RDLC2: The RAMP Model, Compiler & Description Language

Greg Gibeling
UC Berkeley
gdgib@eecs.berkeley.edu

May 20, 2008

RDLC2: The RAMP Model, Compiler & Description Language

by Greg Gibeling

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor J. Wawrzynek
Research Advisor

(Date)

* * * * * * *

Professor K. Asanović
Second Reader

(Date)

Chapter 1

Contents

1.1 Contents

1.2 List of Tables

1.3 List of Figures

1.4 List of Programs

Abstract

Research Accelerator for Multiple Processors (RAMP), is a multi-university collaboration aimed at creating cross-platform architectural simulators for shared-community research, which are orders of magnitude faster than current solutions, primarily by leveraging modern Field Programmable Gate Arrays (FPGAs). The RAMP Description Language (RDL) provides a distributed event simulation and message passing framework, which supports the overall goals of RAMP, including timing accurate simulation through the virtualization of clock cycles to support accurate simulation, and the independent development of structural model simulators, to support collaboration.

In this thesis we briefly set forth the goals of RAMP [9, 90] and describe the RAMP Design Framework (RDF) and RDL which we have created to help achieve these goals. We present two decoupled system abstractions, the host and target models, which are the cornerstones of this work. We document both RDL and the implementation details of the tools, primarily the RDL Compiler (RDLC), which we have written, including its architecture as it pertains to the models and the RAMP goals. We discuss three applications of this work including two from outside of the RAMP project. Finally we discuss both the technical and social aspects of tool building, including several new ideas and features which are designed for integration with the next generation of tools.

Chapter 2

Introduction

The RAMP [9, 90] project is a multi-university collaboration developing infrastructure and models to support high-speed simulation of large scale, massively parallel multiprocessor systems using FPGA platforms. RAMP encompasses a range of methodologies and beliefs about the "best" way to create these simulators. Projects within RAMP range from Hasim [38] to ProtoFlex [28] to RAMP Blue (see Section 15), and are held together by a common interest in using FPGAs for simulation, rather than by a shared base of code or even ideas in some cases. This has proven to be part of the strength of the project over the past three years, as new sub-projects have repeatedly caused the overall objectives to be re-evaluated, and the differences of opinion have led to new ideas and shared learning, particular related to the work we present here.

It is important to note that, while there are a range of implementation and FPGA optimization issues attached to this project, RAMP is not a reconfigurable computing project. Several of the subprojects and the researchers involved are interested in reconfigurable computing, that is using FPGAs to perform computations, but these efforts are separate from RAMP. While RAMP projects will often draw on this work for efficiency reasons, the goal is most strongly *not* to build FPGA-based computers, but to build architectural *simulators* using FPGAs.

The U.C. Berkeley RAMP group has been working primarily on so-called structural model simulators wherein the structure of the simulation mirrors the desired architecture, making the results more defensible, and greatly simplifying the transition from simulation to implementation. We have approached this problem by developing a decoupled machine model and design discipline, together with an accompanying language and compiler to automate the difficult task of providing cycle-accurate simulation of distributed, communicating components. In this thesis, we describe the goals and models of RDF, its realization in RDL and RDLC, several related tools and three applications of our work of varying complexity.

First and foremost the RAMP Design Framework (RDF) must support both cycle-accurate simulation of detailed parameterized machine models and rapid functional-only emulation, in order to allow both meaningful research and debugging. The framework should also hide changes in the underlying implementation from the user as much as possible, to allow groups with different hardware and software configurations, and even those with little to no FPGA experience, to share designs, reuse components and validate experimental results. In addition, the framework should not dictate the implementation language chosen by developers, as widespread adoption is clearly contingent on being able to integrate with the varying FPGA toolchains of the RAMP sub-projects. Taking this a step further, the framework should embrace both hardware and software for co-simulation, debugging and experimental visibility, particularly during the development of complex simulation implementations or evaluation of new architectural features.

RDF is structured around loosely coupled units, implemented on a variety of platforms and languages, communicating with latency-insensitive protocols over well-defined channels. This thesis documents the specifics of this framework including the specification of interfaces that connect units, and the functional semantics of the communication channels. We cover the RDF model (see Sections 3 and 4), description language (see Sections 5 and 6) and compiler (see Sections 8 and 9) in detail.

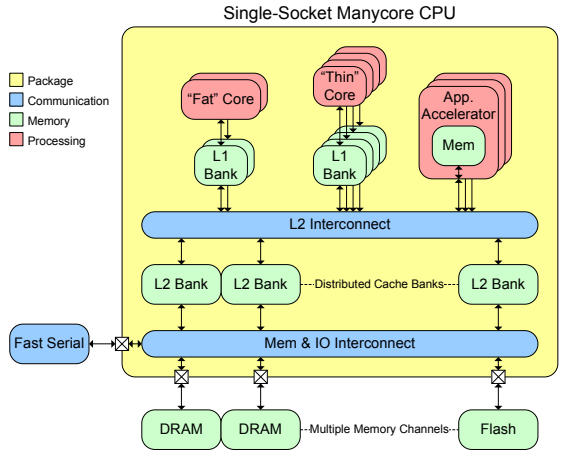
2.1 Problem

There are three major challenges facing CPU designers [74]. First, power has become more expensive than chip area, meaning that while we can easily add more transistors to a design, we cannot afford to use them all at once, giving us a "power wall." Second, memory bandwidth is plentiful, but DRAM latency has been getting much worse, relative to the speed of processor cores meaning we can no longer access more than a small

fraction of memory at once, giving us a "memory wall." Third, CPU designers have become increasingly adept at exploiting the instruction level concurrency available in a conventional sequential program, but the inherent data dependencies mean we cannot execute everything concurrently, giving us an "ILP wall." Together, these three walls constitute a significant problem for computer architecture, and therefore an opportunity for both academic research and industry to bring parallel processors in to play.

Manycore processors, as shown in Figure 1[1], are a natural choice as they are sufficiently easy to construct, well understood and leverage both existing tools, applications and even old Hardware Description Language (HDL) designs to some extent. Manycore processors also have the advantage that their design can be neatly decomposed in to simple processing cores, memory systems and networks, all of which have received attention separately. Most importantly they are easy to scale to match chip sizes, and hopefully performance, by simply by adding cores. Of course such design can still run sequential code by simply confining it to a single core.

Figure 1 Manycore Sketch



Despite all that is known about manycore processors, it is not clear exactly what is needed to ensure that they continue the increase in computing capacity upon which so many industries now rely. The entire computing stack from architecture to application is currently uncertain, at time when rising CAD and silicon costs, multi-gigahertz clock rates and increasing design complexity have made it difficult, at best, to build test chips. This means that at a time when nearly all areas of classic computer science are subject to scrutiny, experimental validation has become all but impossible.

Simulation has long been a bastion of architectural research, providing those without the time or budget a way to test their ideas. Entire research communities have sprung up around simulators like M5 [22], SimpleScalar [10] and SIMICS [14], promoting cross-validation of results and ensuring the continued development of the simulator. Unfortunately, the rule of thumb that concurrency makes application development difficult affects architectural simulators as it does any software, if not more so given their complexity.

At a time when architecture researchers are increasingly compelled to try new and radically different ideas, all the classic methods of experimentation are falling behind. Of course operating system, application and even algorithm developers are in desperate need of test systems to validate their ideas, and are obviously unwilling to put up with second-rate results. In order to close the loop, and ensure a smooth transition from classic single core architecture design to efficient exploitation of concurrency, all of these groups must work together.

> We must all hang together, gentlemen...else, we shall most assuredly hang separately. -Benjamin Franklin

2.2 RAMP

RAMP seeks to provide the tools necessary to allow not only architectural, but operating system, application and algorithm research, by constructing inexpensive, relatively high performance architectural simulations. FPGAs represent a new direction for simulation, promising reasonable performance, price and flexibility, by bringing the efficiency of hardware to bear on the problem of simulating hardware. By constructing a community around shared simulation platforms, designs and tools, the RAMP project will ensure that computer science researchers can cooperate, and hang together.

Because FPGAs can be reconfigured in minutes or seconds, simulations can span a wide range of experimental architectures, without incurring the heavy design cost of Application Specific Integrated Circuit (ASIC) test chips. This flexibility, combined with the efficiency of a direct hardware implementation also puts such designs ahead of high performance SMP- or cluster-based systems which cannot be programmed to simulate a new design, and are non-deterministic, greatly decreasing their believability. Standard software-based simulators fall short on performance, as they do not run fast enough to be usable, other than as a toy, by operation system, let alone application or algorithm

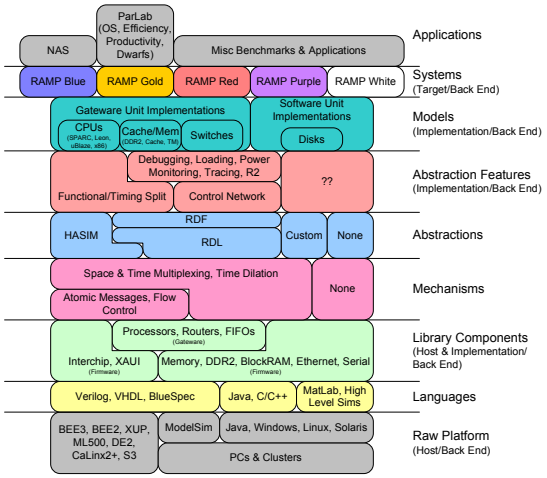
[1] Figure 1 redrawn from original by Krste Asanović.

designers.

Because the goal is to simulate hardware, FPGAs provide an excellent opportunity to leverage design reuse to ensure experimental validity. Simply putting known-good HDL designs for processors in FPGAs results in an instantly believable simulation of the processor in question. Though many new models and designs will need to be constructed from scratch, particularly because known good HDL designs are rarely FPGA optimized, this represents a powerful combination of code reuse and experimental validation with a minimum of effort.

Figure 1 captures most current manycore designs by virtue of being quite general, though when examined more closely there is a very wide range of ideas indeed. Similarly, the researchers within RAMP have many differing ideas about the best ways to construct the necessary simulators, as evidenced by the variety in the lower layers of Figure 2. Underlying these complex differences is the central idea that RAMP simulators must be believable, flexible and interoperable. While there are considerable differences of opinion within RAMP, they are almost entirely about efficiency of simulator construction; the differences are in the methods, not the goals. What's more these differences have provided a constant source of new ideas helping RAMP researchers overcome a wide range of problems.

Figure 2 RAMP Ideas Layering



The goal of RAMP is not to build a single simulator but to find the best way to construct them, and share the artifacts and ideas which underly their construction. Rather than creating a single design, the goal of RAMP is to allow researchers to share their designs. As shown at the top of Figure 2, there are already several example RAMP systems, named by the school colors of the university which is primarily responsible for their design. We will discuss RAMP Blue in more detail (see Section 15), as it has been a main driver of RAMP in general and was co-developed with the work we present.

Given the wide range of ideas and researchers within RAMP, the challenge is to ensure these designs are made portable, interoperable and widely understandable, else they will fail to meet the basic goals outlined above. Architectural research has a long history of custom made tools and one-off designs, neither of which particularly enhances the believability of a simulator or the willingness of even operating systems researchers to adopt it. Adding FPGAs doesn't help, as there are many HDLs and FPGA manufacturers to choose from, not to mention all the intermediate tools and supporting firmware necessary to build and test any reasonable designs. Thus RAMP, for all that it promises to make easy, will require a certain measure of standardization in the form of a unified model, RDF, and automation, in the form of a language and tools, RDL and RDLC, in order to create the community necessary for long term success.

2.3 RDL

In order to provide a standard way to describe RAMP systems, and thereby enable the collaboration necessary to RAMP we have created a framework in which to describe these models, the RAMP Design Framework (RDF), and a language to codify and automate their construction, the RAMP Description Language (RDL).

For believability, both RDF and RDL are focused on structural model simulations, where the structure of the simulator matches the system being simulated. For example, a block diagram of a structural model simulation of Figure 1 would have the same general shape, with the leaf level blocks (such as the cores, or caches) replaced by behavioral model simulations. Such models are easier to construct and more readily believable, as their components can be implemented and verified independently.

In order to build useful simulations it is imperative that we not rely on implementing the system we wish to study, but provide some way to model it. Furthermore, any such simulation must obviously scale beyond the confines of a single FPGA. Automating the virtualization of time and cross-platform support requires some tool to examine a system at the structural level, rather than the Register Transfer Logic (RTL) level of typical HDLs like Verilog or VHDL. RDL therefore provides a high level description of the system being simu-

lated, the system performing the simulation and the correspondence between them.

RDF and RDL support both cycle-accurate simulation of detailed parameterized machine models and rapid functional-only emulation, in order to allow both meaningful research and debugging. Together the framework and language hide implementation changes from the user as much as possible, to allow groups with different hardware and software configurations, and even application researchers with no FPGA experience or interest to collaborate. Of course this includes hiding changes in the underlying HDL, and integration with existing FPGA toolflows. We even support software- as well as hardware-based simulators for co-simulation, debugging and experimental visibility.

RDL is a hierarchical structural netlisting language, designed to describe message passing, distributed, discrete event simulations[2]. RDL is a system level language, and contains no behavioral specification for units, relying on existing design libraries and languages. This is a strength, as it forces RDL to be platform agnostic, and implementation language independent, making it widely applicable and easy to integrate with existing designs.

RAMP and RDF are clearly aimed at large systems where components are created and implemented separately. As such, many of the features in RDL are motivated by the need to tie disparate designs together, in a simple, controllable fashion. Like RAMP in general this document has a bias toward hardware-based simulators, and though RDL in general does not share this bias RDLC2, the most recent implementation, currently does.

RDL provides an abstraction of the locality and timing of communications, enabling timing accurate simulation and cross-platform designs. The biggest benefits of RDL are: deterministic timing, clean interface encapsulation, implementation language independence and the sharing of units between researchers that these things allow.

This thesis describes in detail our first and second specifications of the language for describing such designs, RDL. It it worth noting that this thesis focuses heavily on RDL2 which is significant upgrade, resulting in many additions and several lexical tweaks relative to RDL1.

As it stands, there have been several applications of this language to good effect (see Sections 13, 14 and 15), though some of the murkier details have begun to come to light. It should be clear that this is an ongoing project, and that the list of future work (see Section 16.4) must be taken seriously.

[2]Though it can be used for emulation as well.

2.4 RDF

This thesis documents the RAMP Design Framework (RDF) and the RAMP Description Language (RDL), taking care to explain how they support the goals of RAMP outlined in [90]. However much of this work can be applied more broadly, and in fact the first few applications of RDL have included non-RAMP projects (see Sections 13 and 14). We will postpone a formal discussion of the differences between RDF and RDL (see Section 3.6).

The purpose of RDF is to enable high-performance simulation and emulation of large scale, massively parallel systems on a wide variety of implementation platforms. For the RAMP project, the designs of interest will typically be collections of CPUs connected to form cache-coherent multiprocessors, though this work and RDL in particular are useful for a much wider range of applications (see Section 16.2). In RDF the design of interest, *e.g.* the one being emulated, is referred to as the the target whereas the machine performing the emulation, *e.g.* a BEE2 [25, 37] or PC, is the host.

A target system is structured as a series of loosely coupled units which communicate using latency-insensitive protocols implemented by sending messages over well-defined channels. Figure 3 gives a simple schematic example of two units communicating over a channel. In practice, for typical RAMP designs, a unit will typically be a relatively large component, consisting of tens of thousands of gates in a hardware implementation, *e.g.* a processor with L1 cache, a DRAM controller or a network controller, though they may be as small as an adder. All communication between units is via messages sent over unidirectional, point-to-point, inter-unit channels, where each channel may be buffered to allow units to execute temporally decoupled from each other.

The behavior and abstraction of channel are fundamental to the composition, debugging and cross-platform implementation of the target system. To these ends, channels are loss-less, strictly typed, point-to-point, unidirectional and provide ordered message delivery; in other words channels have the same outward semantics as a standard hardware FIFO, making them easy to work with and reason about. Supporting unit composition despite unknown delay, given the above channel semantics requires that units be latency-insensitive by design. This enables not only composition of units from independent developers, but systems performance research wherein the latency and bandwidth can be varied to determine their impact on performance, independent of any correctness concerns.

Figure 3 Basic Target Model

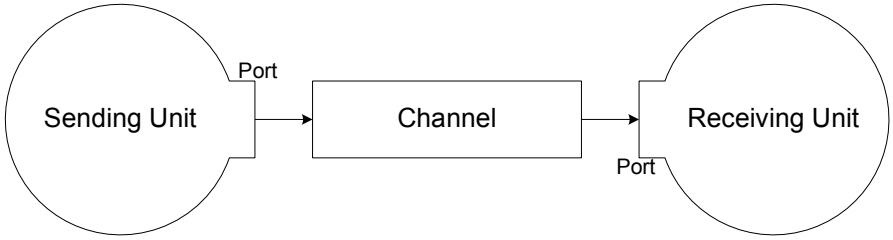
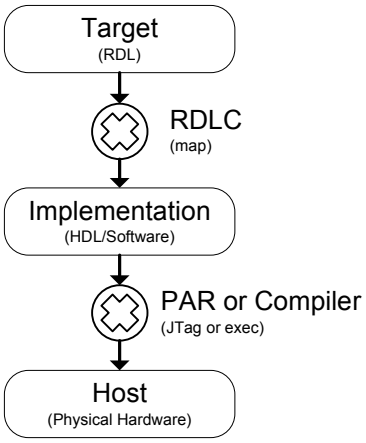


Figure 4 Target, Implementation & Host



RDF is primarily concerned with defining the interfaces and semantics which are required of the target system in order to maintain the goals of RAMP (see Section 3). This in turn will suggest the constraints on the underlying hosts (see Section 4) for, and implementations of these systems.

We make a point to separate the target system, the implementation of that target system and the host for which the target has been implemented, as shown in Figure 4. We do so in order to accurately describe not only the different steps in the toolflow, but to cleanly the design being simulated, the simulator and the system on which the simulator runs. As the separate sections in this document would suggest, we will maintain a very strict separation between the target and host systems, in order to ensure that RAMP target designs will be portable across a variety of hosts. Note that we do not describe implementations separately, as they are merely the per-host, per-target realizations of a particular simulation of a particular target system.

Chapter 3

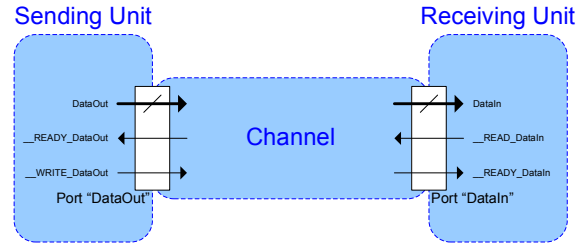
Target Model

The primary goal of the target model is to provide a uniform abstraction of the systems with which a researcher might want to experiment. The target model is an analyzable, standardized model which enables the use of automated tools like RDLC for system building and experimental parameterization. This section describes the target level components of RDF and defines their interaction.

At the target level RDF designs are composed of units communicating over channels, by sending messages as shown both in detail and in a high level schematic in Figure 5. This section expands on RDF (see Section 2.4), including a discussion of units (section 3.2), channels (section 3.3) and the details of their interaction (section 3.4).

Figure 5 Target Model

(a) Details



(b) Schematic

[0]Excerpts from this section have been presented in other publications [48] and are thanks in part to Krste Asanović, Andrew Shultz, John Wawrzynek and the original RAMP Gateware Group at U.C. Berkeley.

3.1 Target Time

RDF and RDL are designed to support a range of accuracy with respect to timing, from cycle-accurate simulations, which are necessary for architectural research, to purely functional emulations, which are useful for *e.g.* application developers who merely need a running system. Purely functional emulations of course represent the simple case, where no measurement of time is required, and any which exists is incidental. However because a simulation will require cycle-accurate results, any implementation must able to maintain a strict notion of time with respect to the target system. Thus we introduce the term target cycle to describe a unit of time, neatly corresponding to a simulated clock cycle, in the target system.

To support multi-clock systems, including GALS systems and those with real-time I/O which together represent the majority of hardware designs, we have defined a unit as a single clock domain, for the purposes of this work. The target clock rate of a unit is the rate at which it runs relative to rest target system. For example, the CPUs will usually have the highest target clock rate and all the other units will have some rational divisor of the target CPU clock rate (e.g., the L2 cache might run at half the CPU clock rate). This implies that two units at each end of a channel can have different target clock rates, complicating cycle-accurate simulation, but allowing this model to encompass a range of extremely important systems. The issue of GALS simulation and other multi-synchronous design points are not considered further in this work (see Section 16.4).

Units are only synchronized via the point-to-point channels. A simulated unit cannot advance by a target cycle until it has received a target cycle's worth of data on each input channel and the output channels are ready to receive another target cycle's worth of data. This forms a distributed, concurrent message passing event simulator, where the buffering (see Section 3.3) in the channels al-

lows unit implementations to simulate at varying target and host rates while remaining logically synchronized in terms of target cycles. The role that target cycles play in synchronization of units is described further in Section 3.4.

As final note, time in the target system is purely virtual, and thus need not be tightly coupled to either real time or the host system's notion of time. The primary goal of RDF is to support research through system simulation, not to build production computing systems. This distinction is particularly important for a FPGA implementations: RAMP is not a reconfigurable computing project.

In an implementation designed for simulation, the target cycle will naturally correspond to a clock cycle in an equivalent non-RAMP implementation, *e.g.* an ASIC design. Having introduced the term target cycle, we now defer a more detailed discussion of time to the following sections, where we will clearly describe what can and must take place within each target cycle.

3.2 The Inside Edge

In order to simplify the implementation of the various units which comprise the target system, we have standardized their interface. In this section we document this interface, called the inside edge, between a unit and the remainder of the target and host systems. Section 3.2.1 lays out the interface and its purpose, and Section 3.2.2 gives the theory of operation.

3.2.1 Description

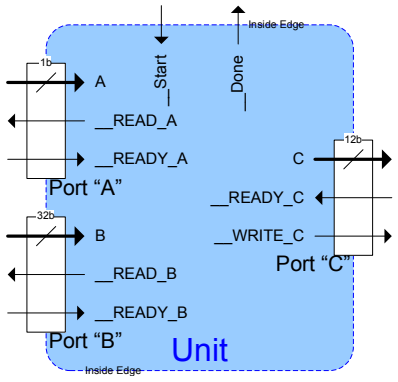
Figure 6 shows a schematic of the interfaces a RDF unit must interact with, for an example unit with two input ports (A & B), and one output port (C). Each port is the interfaces between a unit and a channel (see Figure 3).

While the unit in this example has three ports, in practive a unit may have as few as one port. Though RDLC will gladly compile a unit without ports, there is little point as such a unit can have no interaction with the remainder of the target system.

There are two non-obvious points of interest in Figure 6. First, in addition to the ports, there are two connections labeled `__Start` and `__Done`, which are used to trigger the unit implementation to simulate one target cycle. Second, the ports are each given a message type, which is a simple bitwidth in this example. In general, RDL supports more complex messages through structs, unions and arrays (see Section 5.2).

As Figure 6 shows, each port has a synchronous FIFO style interface, which provides a natural

Figure 6 Unit



match to the channel semantics described in detail in Section 3.3. Input messages are consumed by asserting the appropriate `__Xxx_READ` signal when the associated `__Xxx_READY` is asserted. Similarly output messages are produced by asserting `__Xxx_WRITE`, when the associated `__Xxx_READY` is asserted.

It should be noted that while the above description referred to "signals," which can be "asserted" for the obvious reason that RAMP is hardware centric these constructs can just as easily be represented in software. In software, `__Start` and `__Done` are replaced by a synchronous function call `void __Start()`, which returns when the unit implementation has finished simulating one target cycle. The ports are represented in an object oriented fashion, as a FIFO queue to which the unit object has a pointer or reference.

We do not suggest any connection between target cycles and any unit of time in the software implementation, a fundamental decoupling which should be clear from the description of `void __Start()` as a synchronous function call.

For reasons that will be made clear (see Section 4) we use the term "inside edge" to refer to the interface shown in Figure 6; the collection of the various ports and the two control signals. The basic goal of this interface is to decouple the implementation of the unit (a complex and time consuming task requiring a researcher to write Verilog, Java or similar code) not only from the host, but from the rest of the target system. Complete decoupling is possible through the parameterization of the number and types of the ports (see Section 8.3.2), and the use of platform independant gateware or software for implementation.

In contrast to the term "inside edge" which denotes an abstract interface, the term "shell" is used to denote an implementation of that interface both for a particular language and unit. Unit shells are

typically generated by RDLC (see Section 8.3) and then filled in by an implementor. We have deferred detailed descriptions of both gateware (see Section 8.3.2) and software (see Section 8.3.3) shells.

3.2.2 Operation

Given the above goals, we describe here the functional operation of the inside edge. This description is written in terms of a gateware implementation for conciseness and clarity, not because of any fundamental bias in either RDF or RDL. We make liberal use of the term message which will be more formally defined in Section 3.3.

In gateware the following interaction will occur between a unit and an external entity refered to as the wrapper (see Section 4.2):

1. Before each target cycle the wrapper will present each port with either zero or one messages, signalled by asserting or de-asserting `__Xxx_READY`. This is a key point, as it implies that all messages for a particular target cycle are delivered before that cycle can begin and that each message is delivered atomically, never piecemeal.

2. The wrapper will signal the unit to start a target cycle by asserting `__Start`.

3. The wrapper will wait for the unit to signal that it has completed the target cycle by asserting `__Done`. Note that in software the start/done signaling will be through synchronous function call and return.

4. The wrapper will accept exactly zero or one messages on each output port for which the unit asserted `__Xxx_WRITE` at any point in the target cycle. The wrapper will only accept messages from ports where the `__Xxx_READY` signal was asserted at the beginning of this target cycle. Any attempt to send messages over un-ready ports will result in the loss of said message. Any messages accepted must be delivered in order with respect to other messages sent trough the same port, in accordance with the channel model, as described below. Again, messages are accepted atomically.

The discussion of wrappers (see Section 4.2) includes a complete timing diagram (see Figure 13) for this sequence of events and its interaction with the channel model described below.

3.3 Channel Model

The keys to inter-unit communication, as well as many of the fundamental goals of RDF and RDL, lie in the channel model and the flexibility afforded by its abstraction at the language level. This model ensures that RDF designs faithfully model the performance of the target system. In addition to the inside edge, the channel model is the other main piece of the target model.

3.3.1 Description

The channel model can be quickly summarized as loss-less, strictly typed, point-to-point, elastic and unidirectional with ordered delivery. The remainder of this section covers the details underlying this, and particularly the timing model necessary for architectural experimentation.

Channels should be intuitively viewed as being similar to a elastic FIFO or circular queue with a single input and output, which carries strictly typed messages. In fact, these example constructs will often be the building blocks of channel implementations, also known as links (see Section 4.4.1). From this quick outline we now build upon the basic channel model by describing how they are typed (see Section 5.2), and their full behavior as a component of a target system.

Channels are strictly typed with respect to the messages they can convey. Messages are the flits at the RDL target level; they are the unit of data which a channel carries between units. In keeping with the flexibility goals of RDL, and to expand its utility as a high performance simulation description language, we also introduce the concept of a message fragment to describe the target level phit, the unit of data which a channel carries during one target cycle. Note that none of this discussion affect the use or movement of data within a unit, which is left entirely to the unit implementor.

Channels have several notable characteristics, all of which will be described in detail below.

Type: This includes the widths and types of messages that the channel carries, which must match the declared message type of the sending and receiving ports. In RDL this information is automatically generated based on the ports to which the channel connects.

Bitwidth: The bitwidth of the fragments carried by the channel.

Forward Latency: The latency of fragments, measured in target cycles, from the sending to receiving port. Note that $max(messagesize)/bitwidth$ provides a lower

bound on the latency of the channel. Minimum latency is 0 in RDL but 1 in RDF to ensure that there are no combinational loops, and that gateware may be easily composed by an unskilled user.

Buffering: Indicates the number of fragments that the channel can buffer. Minimum buffering is 0 in RDL but 1 in RDF to ensure there are no zero cycle control dependencies. $Bandwidth = bitwidth$ when $buffering \geq fwlatency + bwlatency$.

Backward Latency: The latency of acknowledgements, measured in target cycles, from the receiving to sending port. Minimum latency is 0 in RDL but 1 to ensure that there are no combinational loops.

As a quick example, expressing the above metrics in tuples $(bitwidth, fwlatency, buffering, bwlatency)$ we can say a 32bit 256 line cut-through (first word fall-through) FIFO with instant acknowledge would be $(32, 1, 256, 0)$.

Figure 7 illustrates the difference between a message and fragment. The channel (represented as a concatenation of registers and an elastic FIFO for reasons which will be clear shortly) accepts exactly zero or one 8bit fragments on each target cycle. Of course it delivers zero or one fragments on each target cycle independent of how many are accepted.

The units, however wish to communicate using 40bit messages. Therefore the messages must be split into 8bit fragments for transport over the channel at a rate of one fragment per target cycle. This means that the sending unit may send, on average, at *most* one 40bit message every five target cycles. To enforce this limit, the `__Xxx_READY` signal (see Section 3.2.1) in the sending unit will remain de-asserted for four cycles after a message is sent, for a 20% port duty cycle, assuming an uninterrupted stream of messages.

Of course the inverse example is equally valid: a message may be smaller than the fragment size of the channel, in which case a message may be sent on every target cycle. However bear in mind that a channel will carry exactly zero or one fragments per target cycle meaning that it may carry no more than a single message per target cycle no matter how small.

3.3.2 Background

Thus far we have explained the interaction of messages, fragments and channels, and in this section we justify the use of this superficially complex model of communication.

Fragments provide RDL with a great deal of flexibility in the definition and performance characteristics of channels. Fragmentation enables the decoupling of the size of messages, which is a characteristic of a unit port, from the size of data moving through the channels. In conjunction with the requirement that units communicate with latency-insensitive protocols, this allows channels to be parameterized with respect to performance without introducing functional incompatibilities with prebuilt units.

Additionally, the concept of fragments is intricately tied to the notion of target cycles. Just as a target cycle is the unit of time in the target system, the fragment is the target level phit, the unit of data transfered over a channel per target cycle. This in contrast to the message which is the target level flit, the unit of flow control. Further discussion of the interaction between time and channels is deferred to Section 3.4.

As listed above and shown in Figure 8, there are four timing parameters associated with every channel: bitwidth, $forwardlatency$, $buffering$ and $backwardslatency$. The bitwidth of a channel (the number of bits it can carry per target cycle) is also its fragment size. "Forward latency" is the minimum number of target cycles which a fragment must take to traverse the channel. "Backwards latency" by contrast, is the time between the receiving unit accepting the message, and the sending unit being notified of that fact. Providing a separate control on forwards and backwards latency allows RDL to capture a wide variety of models. Together the latency parameters allow for highly efficient simulation of Credit-Based Flow Control (CBFC) communications, by abstracting the implementation completely.

Note that the maximum number of target cycles a fragment may reside in a channel before being accepted by a unit is not known, and may vary according to the run-time behavior of the unit. This is the key reason unit must be designed to communicate in a latency-insensitive manner: to allow composition of independently designed units.

The final channel parameter: buffering, is then defined as the number of fragments which the sender may send before receiving any acknowledgement of reception. In general a channel which must support maximum bandwidth communication will require $buffering \geq fwlatency + bwlatency$. However, it is easy to imagine a channel which in fact does not need to be capable of bandwidth equal to its bitwidth, for example a channel somehow carrying exceptions, interrupts or with an otherwise low duty cycle.

At minimum a channel must have a buffering of

Figure 7 Message Fragments

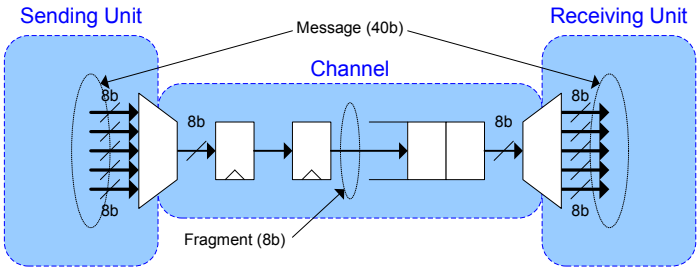
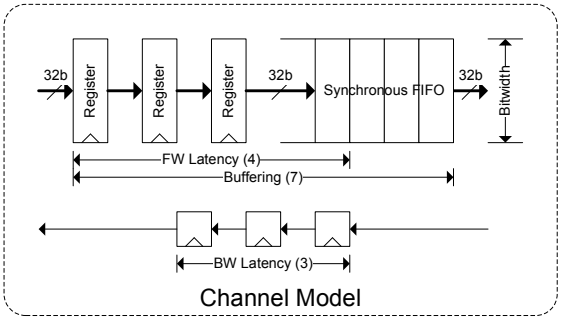


Figure 8 Channel Model



one fragment, in order to ensure that the at least one message may be sent at a time. In RDL, the remaining parameters may all have values as low as 0, however RDF specifies additional constraints for interoperability of separately developed units.

In order to ensure that a target system is feasible to implement and maintain a complete decoupling between units, the forward and backward latency must be at least 1 target cycle. The minimum latency of 1 simply states that the receiving unit cannot receive a message in the same target cycle that the sender sends it. This is required in order to ensure that all messages may be presented to a unit at the beginning of a target cycle (see Section 3.2.2). While a unit may send a message at any point during the target cycle, they are not received until the receiver simulates the next target cycle.

The latency requirement ensures that all data dependencies between units are separated by at least one target cycle, and in conjunction with latency-insensitive protocols this ensures that RDF designs can easily support performance research through parameterized simulation. Note that these requirements apply only to RDF, RDL will accept zero cycle latencies (see Section 6.6) as described in Section 3.6.

3.3.3 Benefits

The benefit of enforcing a standard channel-based communication model between units come from the automatically generated code, as well as design standardization. Users can vary the latency, bandwidth, and buffering of each channel at compile time, rather than design time. The RDL compiler also provides the option to have channels run as fast as the underlying physical hardware will allow, in order to support fast (functional-only) emulation, a useful feature for those building systems, rather than performing simulations. We are also exploring the option of allowing these parameters to be changed dynamically at implementation boot time to avoid re-running the FPGA Place and Route (PAR) tools when varying parameters for hardware simulations.

Development of RDL debugging tools will allow channels to be tapped and controlled to provide monitoring and debugging facilities. For example, by controlling the start and done signals, a unit can easily be single stepped. Using a separate, automatically-inserted, debugging network, invisible to the target system, messages can be inserted and read out from the various channels (see Section 16.4.3).

Finally channels have higher performance than busses (see Section 3.5.1), and are extremely general, making modeling and implementing them relatively easy and inexpensive. In particular the specification of an elastic FIFO model, allows for high performance implementation over standard network and inter-FPGA connections which are the bane of bus and simple wire-based designs. We can easily hide the latency of the host system behind the desired simulation latency, by capturing both with the RDL model (see Section 4.3).

3.3.4 Summary

In this section we have described the channel model and the difference between fragments (phits) and

messages (flits). We have also described in detail the parameters of the channel model, and their interaction with the flow control scheme. This model is the basis of RDF's strength in constructing architectural simulators, as it allows for flexible and parameterizable simulations which faithfully model the performance of the target system.

3.4 Unit-Channel Interactions

In this section we discuss the composability of units and channels and their system wide interaction, especially with respect to simulated time. Up to this point we have only given a broad description of time, units and channels in a target system, though we have even gone so far as to describe, in terms of a possible hardware or gateware implementation, the semantics of the inside edge.

Referring to Figure 9, on each target cycle, the channel will carry exactly zero or one fragments. This restriction is the key to advancing the local target cycle during simulation, whereas during emulation this is a moot point. Time, at the unit level, is advanced upon the receipt of a fragment over each channel, which will make zero or one messages available on each input port. However, in order to advance time in the absence of a message, the channel can be thought of as carrying an "idle fragment."

We have very carefully *not* given first class status to the concept "idle fragment;" it does not appear in the glossary (see appendix B), because there is no requirement that such idle fragments exist. Despite the fact that idle fragments may or may not exist, they are a convenient abstraction to explain the mechanism whereby target cycles advances in the absence of messages. Using this abstraction, we will now explain how time advances and is synchronized in a hypothetical target system.

When the target system, the simulation, is started, each channel might be filled with a number of idle fragments equal to its forward latency. Thereafter on each target cycle, the same port mechanism which handles the fragmentation of messages will generate either a real fragment or an idle fragment, if there is space available in the channel. That is, on target cycles where the unit is not writing to the channel the port (assuming there are no previous messages still being fragmented) will insert an idle fragment to record this absence. The primary reason we do not give first class status to idle fragments is that we can easily imagine situations in which they are not necessary, such as when a channel is directly implemented by registers and a FIFO, or they would be too expensive to send such as over a high latency connection. Timestamps and more direct implementations of the model are among the obvious alternatives.

Aside from forward flow control, back-pressure, possibly implemented using a similar scheme to what is described above, is also an important part of the channel model. Because units can chose whether or not to consume a message on each target cycle, it is possible for a channel to become full. This becomes important as, on each subsequent target cycle the sending unit will not be able to produce a new message. Yet the sending unit will still be told to advance by a target cycle, enabling, for example, the modeling of a non-blocking router unit in RDL.

The composability of units and channels is a requirement in order to ensure RAMP designs can be shared and reused, prompting the creation of the above rules to ensure this composability.

3.5 Limitations

Despite the power of the target model, it does have some limitations, leading to certain tradeoffs both in the target system design and in the choice to use RDL. We have worked to ensure that these limitations are reasonable, and that system designer has a reasonable set of tradeoffs, rather than forcing their decisions.

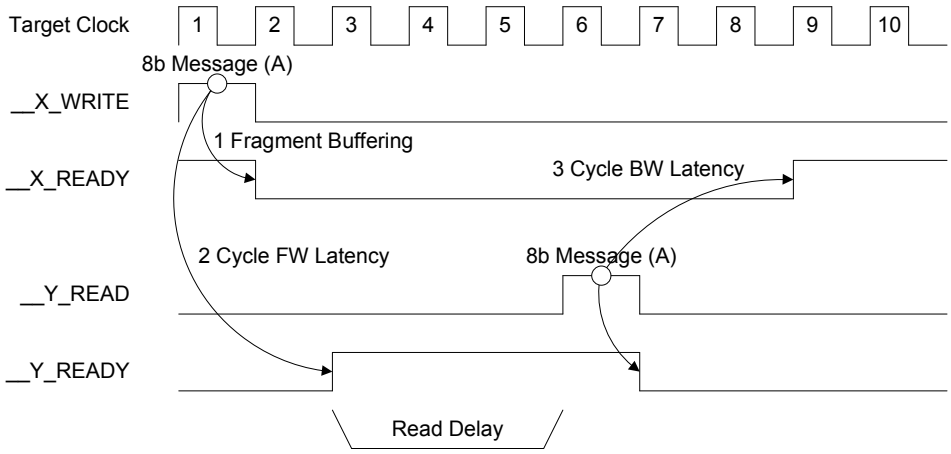
3.5.1 Busses

The target model is not universal with respect to standard hardware design, as it does not provide a native abstraction of busses. Busses were originally a reaction to limited pins counts but are increasingly inefficient in modern ICs and PCBs. To make matters worse, while a Network on a Chip (NoC) or On-Chip-Interconnect-Network design can scale using arbitrary topology, a bus generally has no such options.

The inefficiency of busses has led to interconnects like HyperTransport, PCIe and SATA replacing PCI and PATA, relying on high speed serial communications, rather than time multiplexing, to overcome pin count problems. Even legacy on-chip busses like AMBA are moving towards point-to-point topologies like AMBA AXI [19]. Most damning however is that FPGAs, the primary platform for RAMP designs and therefore RDL, do not internally admit the possibility of busses, but rather implement them using multiplexors and point-to-point connections.

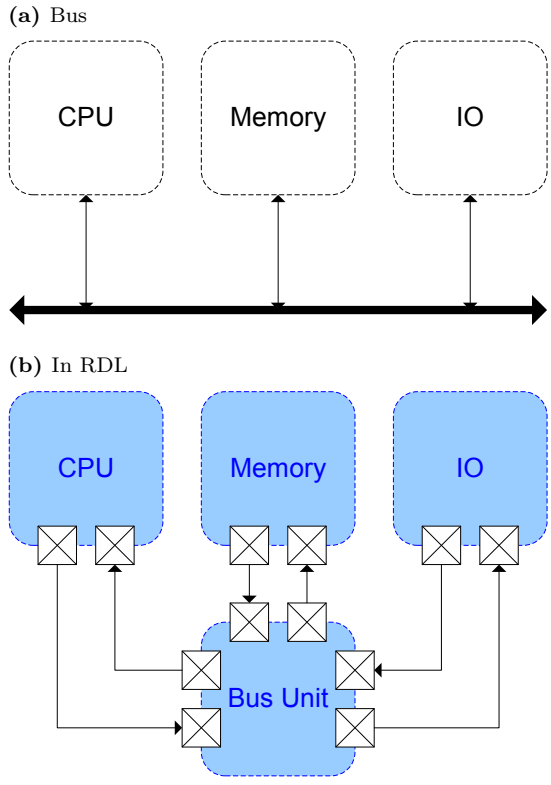
Busses cannot scale, they are inefficient, and are actively being replaced in new designs, all of which leads us to believe that leaving them without a first

Figure 9 Channel Timing



class abstraction is a reasonable decision, in light of the massive automation, simplicity and efficiency tradeoffs to be gained from the simplified channel semantics.

Figure 10 Modeling Busses



On the other hand, the target model should ideally apply to any computer architecture one might wish to experiment with, many of which will include some busses for the forseeable future. There are several strategies for modeling busses. The simplest uses a unit connected to others by channels with 0-latency to represent the bus as shown in Figure 10. This has the drawback of not being RDF compliant, a label which may be bought with the performance penalty incurred by adapting the bus signaling to higher latency channels.

Thus we have two short term solutions for modeling busses, and a long term strategy of avoiding them, which allow the designer a significant amount of freedom all while using RDL, if not RDF.

3.5.2 Cost

RDF and RDL include the ability to perform cycle-accurate simulations on top of a host, such as an FPGA, which has different timing characteristics, a level of abstraction which is not free. Time, chip area and power are all spent supporting this model, both in the implementation of channels and timing control logic. However, the control logic costs are only incurred during a simulation, not emulation, meaning the designer has at least one way to trade cost and functionality.

The majority of the area and power costs are in the channels, and are directly proportional to the timing parameters, and assuming a reasonable compiler implementation (see Section 10.4.1), will be commensurate with the standard FIFOs a channel replaces. The point here is that, while using RDL incurs a cost, this cost is in direct and expected proportion to the structures which the designer specifies, making it hard to call these overhead, but rather simply a cost of the design independent of RDL.

This issue is discussed further in Section 15.1 in the context of CAD tool overhead.

3.5.3 Porting

Finally, there is the problem of porting existing system designs into this model and into RDL, and as opposed to the above overhead costs, the cost of this is in hours of engineering time, a very precious resource indeed.

To mitigate this, there are several options including the simple expedient of turning an entire design into a single unit. While this is generally a five minute task, the downside of the slap-dash approach is that the unit must be monolithically designed, and it precludes cross-platform support, debugging or time dilation, all the main benefits of RDL. However, given that a design implemented this way is likely to stem from a implementation of a target system rather than a model, this may not be particularly problematic, as the system ported this way may not be capable of simulation anyway, but may instead benefit from the automation provided by RDLC.

Of course the reverse is also possible, using an RDL design as a component of some higher level system. Many RAMP designs are likely to be built in something like this manner, particular if there is firmware which must be added outside of RDL (though RDL also allows it to be integrated). In particular, we could envision this methodology being used in conjunction with partial reconfiguration and the BORPH [53, 81, 51] operating system.

In keeping with providing cost-benefit tradeoffs, there is always the option of taking a design as a single unit, and breaking it into separate units as performance and debugging tools are needed along certain channels. Maybe subsystems are built this way, for example a CPU and L1 cache could be two units. Compared to making the whole system one unit this gradual migration allows most of the benefits of using RDL during the migration.

Of course when drawing on existing designs to build units, the simple trick of assigning `__Done = __Start | __Reset` allows us to use an implementation as a model. Clock or register gating will still be necessary for those times when the unit must wait for some external data, but this is an excellent starting for point legacy code. As an alternative to gating, the design could be constructed with 0-latency and 0-buffering channels (essentially wires), though this rather removes the benefits of using RDL and ensures the design does not fit within RDF.

3.5.4 Summary

The primary limitation of the target model is its inability to express busses without a higher level abstraction, an increasingly irrelevant limitation under device and system scaling. The costs are real but are easily controlled by the designer, as the model has no intrinsic overhead, only what is requested by the designer to implement the needed channels and timing. Finally, the process of porting a design to RDL can be simplified as needed, and even for a large design has proven quite reasonable [84].

Thus we have achieved one of the secondary goals of RDF and RDL, allowing the designer to completely control the costs associated with their use.

3.6 RDF vs. RDL

The primary distinction between RDL and RDF is that RDF is a framework of ideas and design methodologies, whereas RDL is a language with a compiler, a seemingly obvious but oft-confusing difference. A design may conform to RDF without reference to RDL of course, but by using RDL, there is a chance to share not only the immediate work, but a common tool-set which serves to remove some of the most mundane work in architectural research. In this section we attempt to differentiate RDL, the language and the tools, from RDF, the framework of ideas underlying it which is slightly more restricted.

The most important theoretical difference between RDL and RDF stems from design restrictions, namely that units must communicate using latency-insensitive protocols. This is vital to the goals of RAMP: collaboration on system implementation and performance research, and yet it does not admit a wide variety of applications which require latency sensitivity. In particular large DSP systems often assume fixed timing in order to obviate the need for control logic.

In RDF, units must not be sensitive to target cycles delays which are external to their own implementation; the units must be latency-insensitive. This is a restriction of RDF, a theoretical restriction, rather than a limitation of RDL or the tools we have developed. The fundamental reason for this requirement is derived from the goal of RAMP to support cycle-accurate simulations without requiring changes to the functional unit implementations. The idea here is that an RDF target system can be configured, by changing the timing parameters of the channels (bitwidth, latency, buffering), to simulate a large performance space. In addition, this ensures that any one unit can be replaced with a functionally identical one, allowing for the painless performance testing of a new architectural component or implementation.

This restriction, while key to large scale systems

design, presents a major drawback of RDF for certain low level projects, many of which aim to use RDL as an implementation or emulation, rather than simulation, language. In order to make RDL useful for a wider range of applications, the language itself is slightly less restrictive than RDF, and makes no such inter-unit protocol restrictions. In point of fact, if one designer takes responsibility for creating a collection of latency-sensitive units, the target system will function accurately, however with the disadvantage that this will heavily complicate compatibility and retard performance research. We leave the value judgment between these to the designer and implementor. This makes the RDL tools both more powerful, and easier to implement as they need not analyze the inter-unit communications, a much harder problem.

By differentiating RDF and RDL, we have thus made the tools simpler and admitted a number of applications (see Sections 13, 14 and 15) while maintaining the restrictions necessary to the RAMP: performance research and collaborative design.

3.7 Conclusion

The primary goal of the target model is to provide a uniform abstraction of the systems which a RAMP researcher might want to simulate. The target model, as laid out above, is an analyzable, standardized model which enables the use of automated tools like RDLC for system building and experimental parameterization.

The above sections specify the complete RDF target model. We have discussed units in terms of their interface, the inside edge, which is composed of a number of ports and certain simulation support signals or functions. We have also discussed channels, their properties and parameters and the difference between messages, the unit of transfer between units and target level flit, and fragments, the unit of transfer over channels and target level phit. The discussion concluded with the details of the interactions between units and channels in terms of the progression of target cycles, the limitations of the target model and the latency-insensitive design requirement of RDF.

This section is deliberately abstract, with the explicit intent of omitting implementation details. This is done to ensure that RDF, and more importantly RDL, have no platform or language bias, and can support cross-platform designs. Though we have discussed most of this work in the context of hardware examples, the work applies equally to software. Elsewhere (see Section 4) we will discuss the details of implementation, including the restrictions on prospective hosts, and the interactions between implementation and abstraction.

Chapter 4

Host Model

One of the main goals of RDF and RDL is to allow the implementation of simulators by mapping target systems to a variety of hosts. To do this it is critical that abstract object, each unit and each channel, be implemented by some physical object. In this section we describe the host model, or the model to which potential RAMP implementation platforms must conform.

Just as units communicating over channels are the basis of the target model, so are platforms communicating over links the basis of the host model, as shown by the green dashed boxes in Figure 11. The primary purpose of RDL therefore is to describe the target, host and a mapping from the units and channels of the former to the platforms and links respectively, of the latter. We will define most of the objects at the host level in terms of the target level objects which map to them, because the host model exists solely to define those requirements imposed on an implementation by the target model, and of course the goals of RAMP.

In comparison to the crisp, rich target model (see Section 3) the host model is far more sparse, with a much smaller glossary. Of course this is one of our goals: to avoid over-specifying implementation platforms and thereby limiting the generality of RDF or RDL.

The target model is idealized in order to create a certain uniformity of design and implementation. The host model by virtue of the fact that it aims to capture the semantics of a wide range of existing systems, is much more pragmatic. The target model is the foundation on which we build simulations, whereas the host model is merely a common-denominator abstraction of all the places we may wish to run these simulations.

Figure 11 shows the eventual cross platform implementation goals. In this section we will define and clarify the constructs needed to realize this, while supporting the target model. A host system is hierarchically composed of platforms (see Section 4.4), time is measured in host cycles, units are encapsulated in wrappers (see Section 4.2), and channels are implemented by links (see Section 4.3). The final constructs, which have no analog in the target system, are the engine (see Section 4.4.2) which drives the simulation, and of course outside world interfaces (see Section 4.4.3).

4.1 Motivation

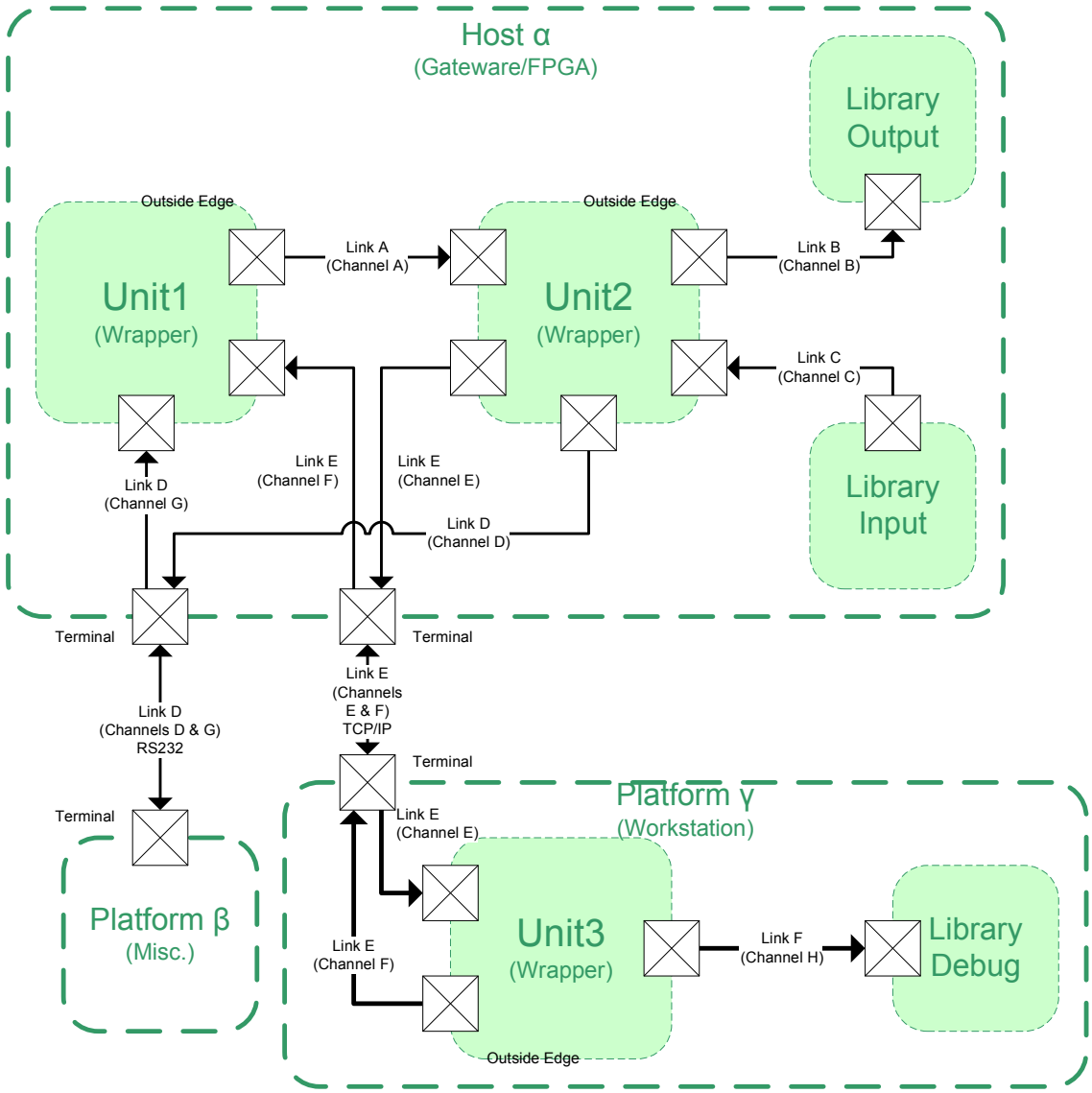
The point of the host model is to allow RDF and RDL designs to span multiple languages and implementation technologies. The host model serves to define the boundaries within which the target model is portable. Figure 11 shows a complex host, with many platforms and numerous links, but it does so without justification. This raises the question of motivation, in particular "why make RAMP designs portable at all?"

In part the portability is required to ensure that cooperative research is not tied to the availability of a particular FPGA board. For example while the RAMP project relies heavily on BEE2 [25, 37] and BEE3 [89, 26] boards, there are universities and even companies which cannot afford a $10,000 board. For these researchers, having the option to run a smaller design on, for example a Xilinx XUP [94] or ML505 [93] or even an Altera DE2 [18] is a valuable alternative. This need is the primary reason why we are clear about the distinction between gateware, which is portable across boards, and firmware which is tied to a particular board.

In addition to board portability, one of the key ideas of RAMP [90] is the ability to assimilate, as much as possible, existing HDL RTL designs. This means that we should, ideally, be able to take a simple processor written in any HDL (Verilog, VHDL or even BlueSpec [56]) and easily create a unit which simulates it perfectly. This goal has proven somewhat elusive due to the varying quality of such code and its lack of FPGA optimization.

[0]Excerpts from this section have been presented in other publications [48] and are thanks in part to Krste Asanović, Andrew Shultz, John Wawrzynek and the original RAMP Gateware Group at U.C. Berkeley.

Figure 11 Host Model



However, for those researchers interested in moving from simulation to *e.g.* ASIC the ability to incorporate the "golden" model of the chip in with an RDL simulation is invaluable.

There is also the perception (and reality) of start-up costs. If one is asked to learn both RDL and a new HDL (*e.g.* VHDL when they already know Verilog) the price of using RDL may well be too high, particularly at universities where graduate student time is at a premium.

In sum, the need to integrate existing HDL means that RDL must be language independent, or else the cost of integrating these designs becomes prohibitively high.

Thus far we have justified the need for both board and language level portability. However, RDF must go one step further and, though seemingly contrary to the stated goal of RAMP [90], embrace abstraction portability, to cover both hardware and software simulators.

Researchers have been designing software-based architectural simulators for many years [14, 22], and to great effect. While these simulators are prohibitively slow for software development and difficult to parallelize, they are functionally correct and highly believable. By allowing RDF to span hardware and software we allow hybrid simulators [28] to be built using the speed of hardware where possible, and the flexibility of software where needed.

Even more interesting, by allowing the partition between the two to move easily, we can allow a researcher to prototype their ideas in software and

run with nearly hardware speed, in essence enabling co-simulation of the simulation. Given that RAMP must reach out to researchers who have no experience or interest in FPGAs, this is likely to be a vital first step. It allows the validation of the reseacher's ideas, while at the same time showing some of the benefits of using hardware to simulate hardware. Should their ideas prove promising enough to warrant further analysis, it could then be migrated to a gateware implementation for higher performance, validation by colleagues and use by operating system or application developers.

Of course there are also practical matters behind the support for software. Debugging a new unit can be made easier by comparing gateware and software models for the same functionality. A software unit can have access to OS level services, such as files and network access, making it easy to load data in to an experiment and capture the results. Of course a software unit might also be something as simple as a HDL simulator (*e.g.* ModelSim) simulating a gateware unit, for higher visibility.

With portability comes variety, thus the second reason for the host model, is to allow automated tools (RDLC) to digest information about the host on which a target is to be implemented. Thus in addition to simple things like what language to generate code in, the host model exists to capture such details as how to simulate channels efficiently, and how to move data between platforms. The motivation for the host model stems from the need for portability in conjunction with the desire to manage this portability as automatically as possible.

4.2 Wrappers

In order to isolate the unit implementation, which should be written in gateware or portable software, from both the underlying platform and the units surrounding it, we encapsulate units in wrappers. Figure 12 shows a unit (see Figure 6) expanded to include the wrapper, links and control mechanisms required to implement the target model (see Section 3). The wrapper is the container for all of the implementation details required to support the inside edge interface on top of the underlying platform.

We introduce the term "outside edge" to describe the interface between the wrapper and the links, as well as the rest of the implementation. The fundamental job of the wrapper is to support the inside edge interface in the requested implementation language and translate it to the outside edge interface. To this end, the wrapper will need to contain a significant amount of functionality, as described at a high level in the below sections (except for the link interfacing logic, which is described in Section 4.3).

Note that while we present this functionality as being implemented inside the wrapper, this is a somewhat negotiable point. While the functionality is all necessary, in the name of efficiency, individual links may sometimes implement this functionality rather than the wrapper.

4.2.1 Marshaling

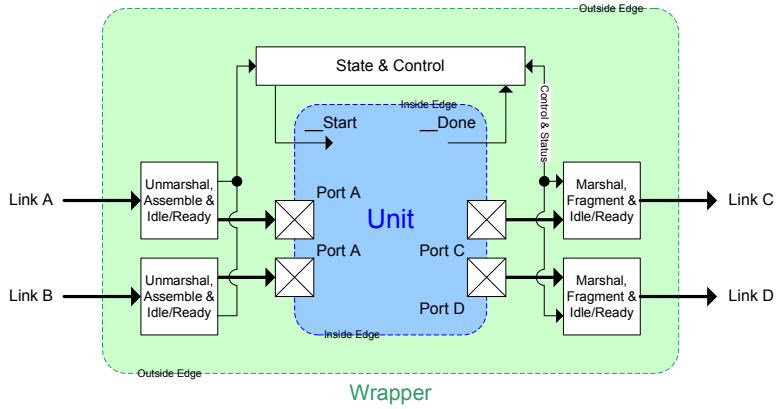
RDL includes language level message constructs: array, struct and union (see Section 5.2), to support the construction and debugging of more complex target systems at a higher level of abstraction, as well as integrating with languages like SystemVerilog, While arrays and structures of bits pose no particular difficulties during conversion to a flat bit-vector for transport, tagged unions are more complex. A union of two types of different length, such as a bit and double-word, means that the resulting data can be either 2 or 33 bits in total (including a 1 bit tag). Given that message types can be arbitrarily nested and sized, this can quickly lead to a large disparity between the maximum message size, and the average message size and thus introduce serious inefficiency particularly over longer latency links and channels.

Take the example of a channel from a CPU to a memory, which may carry a read request message (32 bits) or a complete cache line write ($256+32=288$ bits). If this channel is to cross a inter-platform boundary over a low performance link, and particularly if writes are relatively rare in our example, then the overhead of transmitting the extra bits is likely to become a significant bottleneck.

As an efficiency measure, therefore, the wrapper is responsible for marshaling complex messages, reducing their size by removing the bits which are designated irrelevant to this particular message. In the process the wrapper is responsible for calculating the size of the resulting marshaled message so that the link knows how much of it is valid. Thus marshaling is primarily responsible for dealing with union message types with variable bit sizes, and is trivial for other messages.

Of course unmarshaling is the opposite process, of adding garbage data to expand a marshaled message back to its original, canonical representation, and requires only the union tags, not the computed message size. The exact code which is generated for both marshaling and unmarshaling, is discussed elsewhere (see Section 8.4). Note that this logic may be single cycle, pipelined or multicycle as needed to optimize a hardware implementation as described in Section 4.2.4.

Figure 12 Wrapper



Note that marshaling is not free, either in space or time, and its use should be balanced against the benefits provided. As such RDL must, and does provide the designer with control over the use of marshaling (see Section 5.4.5). Given the relative simplicity of these operations, it should be possible to automate the decisions about how much marshaling, if any, to perform on a per-channel basis (see Section 12).

4.2.2 Packing

packing is necessary to multi-dimensional array messages in languages, like Verilog, which have no such native support, just as marshaling is necessary to support union-typed messages. Packing in contrast to marshaling, requires no active logic in most HDLs, and instead consists of cleverly renaming wires. Of course in software, packing may be as simple as providing a custom abstraction of message arrays. Packing, which is logically part of the wrapper, may actually appear inside the unit shell depending on the restrictions of the implementation language.

4.2.3 Fragmentation

Before transmission over a link, messages must at least be logically decomposed in to fragments or fragmented in order to support the channel timing model (see Section 3.3). This may include physically decomposing the message in to fragments, or it may include nothing more than dividing the size of the message calculated during marshaling (see Section 4.2.1), by the bitwidth and recording this value for time accounting. The opposite operation, reconstructing the message, again while honoring the timing model, is called assembly.

While the actual fragmentation and assembly are logically the job of the wrapper, in some implementations they will be part of the link for efficiency reasons (see Section 8.4). Even in this case the wrapper is responsible for converting any control and data formats necessary to send the message over the link, for example converting from an abstract bit-vector to an array of bytes in a software implementation.

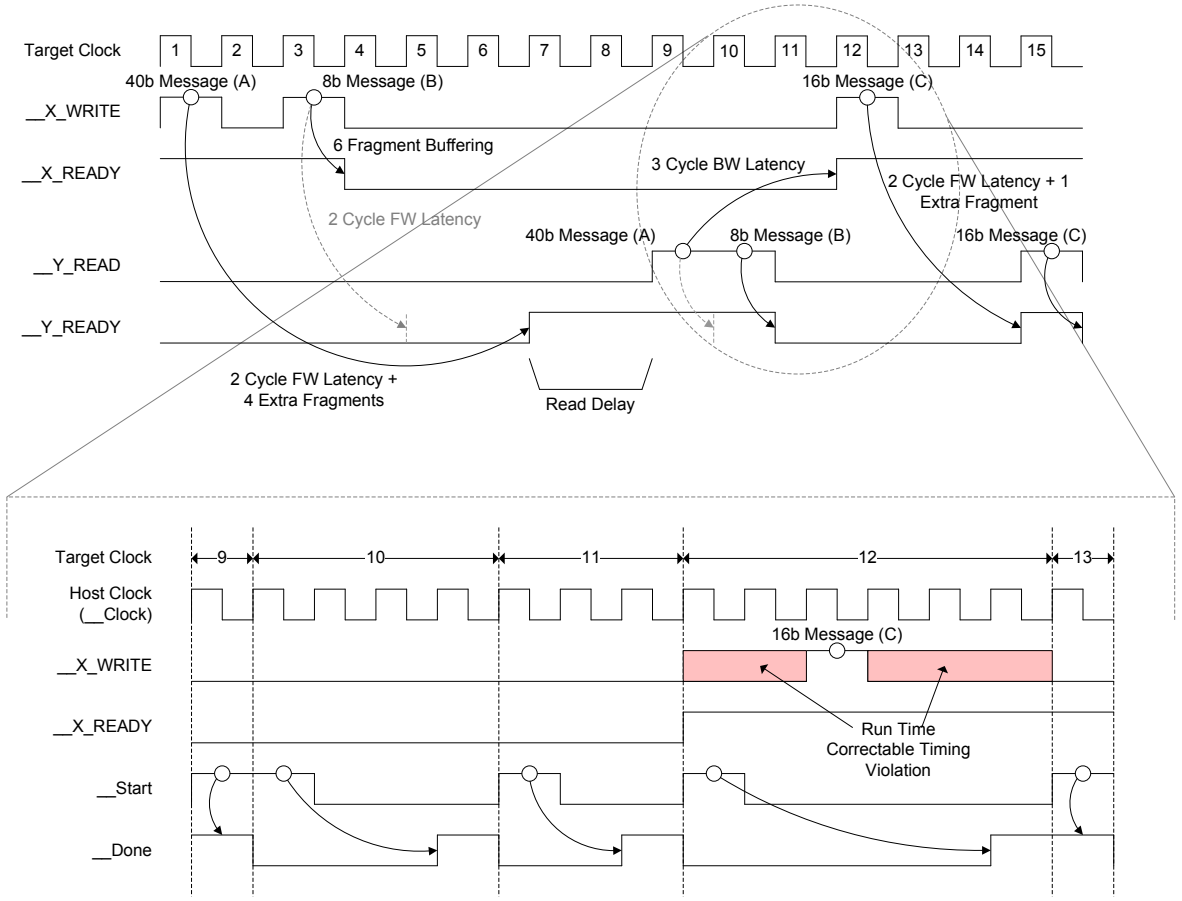
4.2.4 State & Control

The other main duty of the wrapper, in addition to massaging the messages properly, is to control the advancement of simulation time. This is one of the most important features of RDF simulations, as the separation of simulated target cycles from host cycles is the basis of all space-time tradeoffs in the simulator. These tradeoffs might range from simply pipelining a unit implementation rather than parallelizing it, to migrating unit implementations between gateware and software for development or debugging. Figure 13 shows the expansion of target cycles to multiple host cycles (see Figure 9).

Whether it is called “time dilation” or “virtualization of time” the point is that target cycles are decoupled from host cycles. Experimental validity demands that RDF simulators match real machines cycle for cycle, and it is the job of the state and control logic to ensure this by tracking target cycles. This logic provides for cycle-accurate simulation on vastly different implementations by implementing a distributed event simulation, where the events mark the unit-local simulation of target cycles.

Primarily the state and control are responsible for determining when a target cycle may begin based on incoming fragments (including idles or timestamps, see Section 3.2) and generating whatever implementation of “idle fragments” is appro-

Figure 13 Host Timing



priate (see Section 3.4). Given that individual links are responsible for reporting their readiness using the interface shown in Figure 14, this consists mostly of waiting for the unit and all the links to be ready (by examining the `__Xxx_STALL` signals), pulsing `__Start` and waiting for `__Done` or simply calling `void start()`. However, marshaling, packing and fragmentation may also incur host cycle delays, which must be accounted for.

While data dependent timing complicates the implementation it allows for powerful abstractions, such as the implementation of a cache model with no actual storage inside of a single unit. In this example the cache tags could be stored in DRAM alongside the actual data. A memory access through this simulated memory hierarchy would them consist of at least two DRAM accesses, one for the data and one for the tags. In order to maintain target cycle accuracy the memory hierarchy unit could simply return the responses on whatever target cycle dictated by the value of the tags.

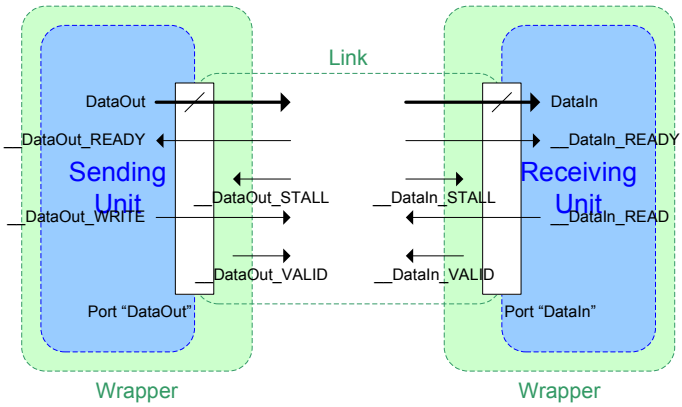
The state and control logic are also responsible for the enforcement of channel semantics, including ensuring that no more than one message is read or written through each port on a given target cycle (see Section 3.2.2). In addition this logic must ensure that message transmission and reception is atomic, despite the fact that message delivery, because of fragmentation, is not. For example in Figure 13 there are two red areas at the host level showing that while the unit implementation might attempt to send two messages in one target cycle the wrapper can easily catch this, ignore it or report an error as desired.

Of course, the firing conditions will also depend on the automatic debugging functionality, which in many cases will include the ability to pause or single step a unit [24, 23]. The main rules for firing are outlined elsewhere (see Section 3.2).

4.2.5 Summary

Wrappers, because they are are conceptually simple but with many variations, will be automatically generated by RDLC (see Section 8.4). This is possible because while wrappers are implementation language dependent, they require no design information beyond basic link parameters, and the RDL

Figure 14 Host Handshaking



description of the target system. Note that there are several optimizations which can be enabled or disabled at this level, a set of decisions currently left to the RDL programmer, though they should eventually be automated (see Section 12).

4.3 Links

The fundamental purpose of RDF and RDL is to connect units which wish to communicate, meaning that a proper abstraction of the available host level communication media is vital. Links are the host level analog of channels, just as wrappers are the host level analog of units. The term "link" is used to denote both the raw physical resources which connect platforms (the link), and the implementations of channels (a link implementation or instance), a subtle dichotomy which may at times seem confusing.

In order to allow designs to be split arbitrarily between platforms, without regard to correctness, links may be multiplexed. Thus while a synchronous FIFO with some timing logic implementing a single channel within an FPGA would be an excellent example of a link, an an RS232 cable connecting two computers running two halves of a large software design would be equally valid. Links which connect units mapped to a single platform are generally expected not to be multiplexed, where as links between platforms will often by shared (see Section 4.4.1), though neither of these are required.

Unlike channels, RDL and RDF impose almost no restrictions on links, other than their ability to support the channel model (see Section 3.3), thus links implementations can be built on by nearly any data transport. Examples include direct gateware implementation using registers and FIFOs, software circular buffers, high speed serial connections, busses, or even UDP/IP packets. Of course not all of these implementations lend themselves to direct channel implementation, for example networks such as Link E in Figure 11 are often lossy, and unordered and might require TCP to provide the required guarantees.

One major difference between links and channels is that links needn't be point-to-point, and may include switched networks or busses. This is because with links, unlike channels, there's little need for the language or tools to capture the operational details, for optimization (see Section 12), debugging or even code generation, as explained below.

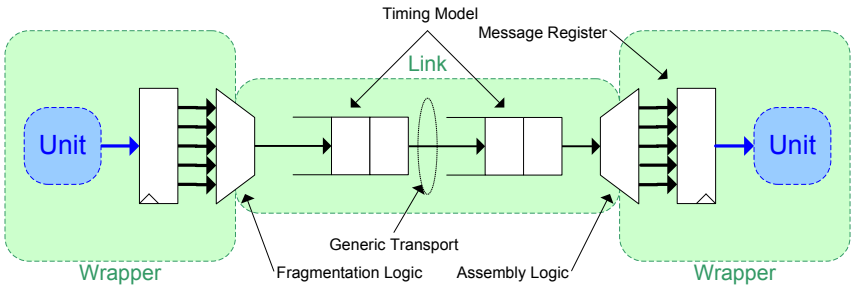
While the link model admits the possibility of a network-based implementation, it remains faithful to the point-to-point channel model. To be more specific, while links may be any topology, including relying on routed networks, this functionality is not exposed even at the level of the implementation.

4.3.1 Generation

Link implementations will often include functionality which is highly platform, language and transport dependent, unlike wrappers, which can be generated uniformly and automatically. Thus while some links (such as circular buffers in software and FIFOs in hardware) will be natively supported by RDLC, others like TCP/IP connections will require pre-written library components and compiler plugins. We use the term "link" to refer to a particular physical transport, and "link generator" to refer to such a compiler plugin, which we will discuss elsewhere (see Section 10.4.1). By abstracting links behind a general compiler interface, rather than a language or model construct, we allow significant flexibility in implementation methodology and simplify the compiler without loss of generality, as links need not be analyzed.

A link implementor is responsible not for im-

Figure 15 Link



plementing the complete channel model, much of which is implemented in the wrapper, but for implementing data transport. What is important is that links provide loss-less and in-order delivery of data. Of course these attributes may be implemented by retransmit and sequence numbers, or other such similar mechanisms.

Aside from inter-platform links, such implementations as asynchronous FIFOs, are useful for *e.g.* making clock domain crossings at the host level completely transparent to the target system. It is our goal to include several standard link generators with the RDLC tools (see Section 10.4.1) both for users and as reference designs for those wishing to implement new links.

4.3.2 Implementation

For low performance links, credit-based flow control is a more natural and efficient implementation of the channel handshaking than the abstract crutch that is "idle fragments" (see Section 3.4). Credit-based flow control will happily account for the latency both in the data transfer (fragments moving forward) and the handshaking (credits moving backward). At startup, the sending unit would be given a number of credits equal to the buffering capacity of the channel, thereby allowing it to send that many fragments prior to the receipt of any additional credits. Of course the receiver should return credits to the sender, as it consumes fragments, thereby freeing buffer space. Because it will take $fwlatency$ cycles for the fragments to reach the receiver and another $bwlatency$ cycles for the new credits to reach the sender, the channel will require $buffering \geq fwlatency + bwlatency$ to achieve $bandwidth = bitwidth$.

An alternative implementation would be a distributed FIFO with each stage separated by one target cycle of latency, both for data traveling forward and flow control traveling backwards. Because both flow control and data are subject to latency, each stage will require two fragments worth of buffering, providing for the same $buffering \geq 2 * latency$ required to reach $bandwidth = bitwidth$.

Because most links will already implement some form of credit-based flow control, significant efficiency can be gained by re-using this to implement the channel flow control (see Section 3.3). In particular the host cycle latency of the link should be hidden behind the target cycle latencies of the channel. Note that this is unlikely to save much area (memory or FPGA registers) for the simple reason that even the lowest performance link will sometimes be fast relative to the simulation, meaning the timing model components must be prepared to buffer the maximum amount of data. In particular, without dataflow analysis which is prohibitive given the lack of unified unit implementation restrictions or even a common language, there is no way to know when and how often a unit may be paused between target cycles, thereby requiring its input channels to use their full buffering space.

4.3.3 Zero Latency Links

RDL admits channels with latency (forward or backward) of 0 target cycles (see Section 3.6), even though RDF disallows them. This restriction on RDF has theoretical and practical underpinnings at the research level, but also at the implementation level. While implementing a 0-latency link in software is easy because all target cycle accounting is artificial, it is quite difficult in hardware.

While two units on a single FPGA platform, which share a clock domain can easily be connected by a 0-latency link (a wire) there may be no such link between two FPGAs. Worse yet, imagine a unit implemented on an FPGA connected to a unit implemented in software, where latency is necessarily an order of magnitude higher.

In these cases, a simple link such as a wire is not possible. However, some simple guarantees about the nature of the signaling over the channel are enough to make a 0 target cycle latency possible. In particular if the number of signal transitions or

messages has a finite upper bound, and the transitions are synchronous to the host clock, or otherwise detectable by the wrapper, they can be faithfully transmitted. The finite upper bound on the number of signal transitions allows the wrappers to positively identify the end of the target cycle. Inter-cycle transmission merely requires that the sending and receiving wrappers be modified to violate the "at most one message" rule of ports.

Note that while these links are technically possible, and may have limited use during debugging and development they are strictly prohibited by RDF. There is little RDL support (see Section 6.6) for these links, in part to help remove the temptation to design non-portable non-reusable gateware, which is contrary to the goals of RAMP.

4.4 Platforms

Stated simply a host system is merely a connected collection of hierarchically defined platforms. Whereas wrappers are the host level analog of units, platforms are the physical resources on which both wrappers and units are implemented. We have also found it helpful to add the distinction between front end, which might be a monitoring or FPGA programming machine, and back end[1] which consists of all the platforms on to which implementations are mapped. This distinction becomes important when discussing complete RAMP experimental setups, which include monitoring, debugging and time sharing of the experimental resources, particularly expensive FPGA boards or compute clusters.

In the remainder of this section we cover the features which platforms have. We begin with links and terminals in Section 4.4.1, as these are the primary model features. We go on to discuss engines, which are responsible for driving the simulation by providing scheduling and other such basic facilities. Finally, we discuss input and output, which exist mostly outside of the host model and represent the necessary "escape hatch" from this model.

4.4.1 Links & Terminals

Terminal are the points of connection which adorn the boundaries of platforms, and through which all inter-platform communications take place. Whereas links are the host analog of channels, terminals are the host analog of ports. Shown at the top of Figure 16 is an example of two platforms, each containing a single unit and each with a single terminal, which have been connected to carry the channel between units. This figure may be contrasted against Figure 15 in that it shows a higher level schematic, and an example of a link spanning more than one platform.

While links have been discussed extensively in Section 4.3, they were discussed in the context of implementation rather than use. This section discusses links and terminals as they are used connect platforms to form a complete host.

Figure 16 shows the implementation of a channel mapped to a link which connects two platforms at various levels of abstraction. The ability to create such distributed simulators is an obvious and base requirement of RDF, as the RAMP project desires simulations of 1000 core machines, which given the size of even the smallest cores and largest chips will take multiple FPGAs. As an example RAMP Blue [63, 64] took 21 boards to reach 1000 cores. There are ideas such as implementation multithreading to help reduce this, but scaling trends will always demand that we span multiple platforms, in order to stay ahead of processor implementations we seek to experiment with.

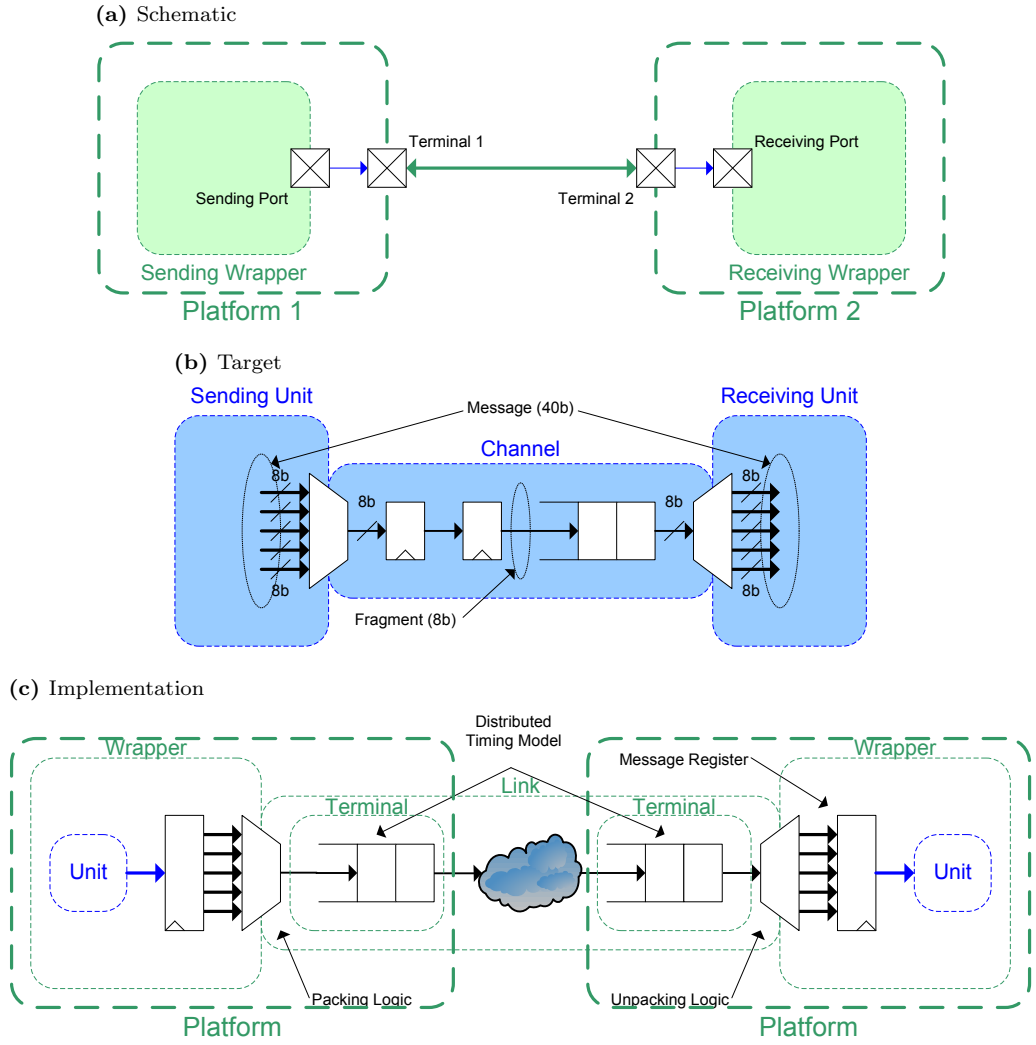
Terminals, like ports, serve to define the type of the links to which they are connected. Unlike ports, whose type derives from the messages they communicate with, the type of a terminal is a more nebulous concept, often being highly implementation specific even if platform independent. Terminal types, according to this model, are merely opaque identifiers which can be compared for equality, though like ports complex terminal types may be built from simpler ones using arrays, structs and unions (see Section 5.2). This allows us to make assertions such as: all of the terminals connected by a link (remember links needn't be point-to-point) must be of the same type, without regard to what kinds of link types a researcher may wish to use.

It the future, as performance demands it is entirely possible that performance information may become a part of the terminal and link meta-information captured by this model. We believe this would facilitate the creation of a variety of tools designed to automatically route signals between platforms in the presence of multiple links (see Section 12).

It is also valuable to note that while, as stated above (see Section 4.3), links may be implemented using some kind of routed network, this is not exposed even at the level of the host model. In particular an implementation includes a mapping from channels to links, but includes no specification for routing or connecting links together end-to-end. Of course a particularly enterprising link implementor

[1] Not to be confused with a compiler's front and back ends.

Figure 16 Cross Platform Link



(see Section 16.4.5) could create an abstract link which connects all platforms in a host to each other as shown in Figure 17. In this example a physical ring topology is abstracted by a host level network, effectively creating a single link connected to four terminals, one per platform.

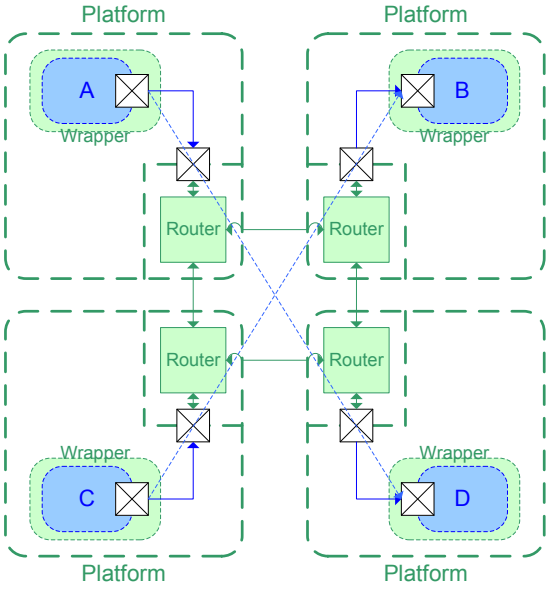
While such host level network links are easily captured by this model, it is not the goal of this model to create this an abstraction. In particular this abstraction is likely to be extremely expensive on FPGA platforms, and discourages the exploitation of platform locality, which would allow much more efficient implementation.

4.4.2 Engine

All implementations will require some way to drive them, *e.g.* a clock in hardware or a scheduler in software, in addition to unit implementations and links. The exact job of an engine is to decide when and which wrappers are allowed to run, often in conjunction with the wrappers themselves, which manage local firing rules.

In hardware, this task often reduces to providing the reset and clock signals required for a simple synchronous design, thus keeping the engine simple. The engine for HDL simulation (*e.g.* using ModelSim) would use behavioral constructs to generate a clock signal out of thin air, as it were. Engines for FPGA platforms by contrast are responsible for any clock processing logic, and generating a reset signal, either from programming control signals or from an external button (see Section 10.4.2). Note that these engines will range from simple wires for signal generated by firmware to complex circuits in and of themselves, particular if a platform is to be integrated with debugging based on clock gating

Figure 17 Host Level Network



[23, 24] or the like.

In software the engine is effectively a user level scheduler, where each wrapper must be run, and the engine must decide which ones to run, when and even where in the case of a simulator running on a multiprocessor or distributed computer. Any algorithm for scheduling a dataflow graph will be suitable for scheduling units, a topic we leave to the implementor. RDL and the compiler both admit the possibility of a wide variety of schedulers each with different policies, by the simple expedient of not restricting the decision at all. This has led to an interest in supporting efficient software emulations of such projects as Click [62] and P2 [69, 68] using RDL (see Section 14).

The nature of the engine, the similarity to software-based simulation schemes is what leads us describe RDF designs as "distributed event simulators implemented in hardware."

4.4.3 I/O

At some point any simulator will require interaction with the outside world to access input data and deliver its results. Through our description of the host model, we have focused on the components of a host system which are necessary to implement a target design because RDF and RDL have no native abstraction of non-link data transfer, and no concept of non-channel data traveling over links. Instead, I/O is treated as an integral, though second class concept which is platform dependent. There are currently no components of either the host or target models which offer an abstraction of I/O.

RDL relies on compiler plugins (see Section 10.4.4) to provide this functionality, while RDF is silent on the matter. The decision to omit I/O from the primary platform model has proven a good one for our initial work as we had no experience to guide any decisions in this area, though it should be re-evaluated once more applications are available (see Section 16.4).

One issue which naturally arises in this context is the concept of a unit without channels, existing solely for some implementation specific code. We consider this to be contrary to both the host and target models as it conflates the simulator with the system performing the simulation, reducing the generality of the latter to no gain. Instead the preferred implementation path would be for such code to exist as firmware, viewed as part of the platform and added in to the code base as part of the final compilation or linking steps after RDLC.

4.5 Conclusion

The host model, the model to which potential implementation platforms must conform, has been designed to capture a wide range of physical platforms on which the RAMP project will want to implement simulators. We have discussed in some detail the exact nature both of these hosts and the mapping of a target system to a particular host to create a simulator implementation.

At the core of the host model, a host system is composed of a series of hierarchically defined platforms connected by relatively arbitrary links. A mapping from a target to a host then consists of a correspondence between units and platforms, and channels and links. Units are then encapsulated in wrappers, which support the inside edge interface based on the outside edge interface between the wrapper and the relevant links and engine. While the wrappers are generally uniform, the links are arbitrary and mostly opaque to this model allowing them to be quite general.

Chapter 5

RDL Statics

In order to provide a standard way to describe RAMP systems, and thereby enable the collaboration necessary to RAMP we have created a framework in which to describe these models, the RAMP Design Framework (RDF), and a language to codify and automate their construction, the RAMP Description Language (RDL). RAMP and RDF are clearly aimed at large systems where components are created and implemented separately, as such, many of the basic features in RDL are motivated by the need to tie disparate designs together, in a simple, controllable fashion. RDL is a declarative system level language, and contains no behavioral specification for units, relying on existing design libraries and languages. The RAMP Description Language (RDL), has been designed both to help formalize RDF (see Sections 3 and 4), and to capture systems within them for manipulation by automated tools, in particular the RDL Compiler (RDLC) (see Sections 8 and 9).

In this section we discuss the static elements of RDL, which are essential to statically, or "at compile time" describing these design libraries, languages and constructs, the basic structure of RDL descriptions and type declarations. Section 6 covers the netlisting aspects of RDL, through which target and host systems are constructed. This section focuses on the the lexical structure, syntax and semantics of RDL particularly as they relate to the goals of RAMP (see Section 2.2).

5.1 Basic RDL

This section is an introduction to the lexical structure and syntax of RDL and those constructs which are necessary in any language. This section gives the basic language structure of RDL, versions 1 & 2 though with an emphasis on RDL2. Note that at the time of this writing the most recent releases are 1.2006.3.1 and 2.2007.8.13, wherein one can read the major version number and date of the release.

5.1.1 Literals

Any and all numbers in RDL may be specified in base 2, 8, 10 or 16. A number which starts with a digit 1 through 9, is interpreted as decimal. A number which starts with a 0, must contain a base indicator (`b` for binary, `c` for octal, `d` for decimal or `h` for hexadecimal) followed by a number in that base. Thus the number ten could be written in a myriad of ways: `10`, `0b1010`, `0c12`, `0d10` or `0xA`.

Numbers currently have several implicit limitations. First numbers are internally represented with Java integers, making them implicitly 32bits and signed. Second, the inability to explicitly associate a bitwidth and arbitrary base for numbers is a definite limitation.

In addition to numbers, many parts of RDL use string literals. Not to be confused with identifiers, covered below, string literals are essentially opaque to the compiler, and are general used in conjunction with plugins or parameters to unit implementations. Strings in RDL are always surrounded with double quotes, and the only escape characters recognized are `"\n"` and `"\""`. Any other escape characters will be included literally, meaning for example that `"\g"` would be identical to `"g"`.

5.1.2 Identifiers

Identifiers or "names", in RDL as in many common languages, are case sensitive and must start with an underscore or a letter though they may contain underscores, letters and numbers.

In RDL there are two kinds of identifiers: static and dynamic. Static identifiers are so called because they name objects in the static RDL namespace hierarchy, that is objects which appear in RDL source text such as units, platforms, maps and types (message, port and terminal types, see Section 5.2). Dynamic (see Section 6) identifiers on the other hand name objects in the host or target systems described by the RDL, that is to say objects in the unit or platform instance hierarchy. Program 1, which we will explain later, only contains static

identifiers, since there are no actual unit, platform or map declarations and thus no instances.

Static identifiers specify declarations within a static, that is known at compile time, scope and dynamic identifiers specify instances within a dynamic, or runtime scope. However, it should be noted that because RDL is meant to describe simulations of hardware to be implemented in hardware, even the so-called dynamic structure of the target and host must be elaborated at compile time.

Those familiar with C++ will have no trouble with this distinction, partly because in RDL as in C++, compound static identifiers are separated with `::` and compound dynamic identifiers with `.` (`->` is also used in C++ of course). Note that this in contradiction to Java where both static and dynamic identifiers are separated with `.`, and standard HDLs which historically have no concept of a compound identifier and barely any concept of scoping. Lest the analogy to object oriented languages confuse the matter, we reference them merely as examples of lexical structure, RDL is not object oriented and has no inheritance mechanism.

The difference between static and dynamic identifiers will become more clear by way of examples in the following sections.

5.1.3 File Structure & Declarations

RDL descriptions are organized as a series of declarations of various kinds within a simple file, similar to the file organization in C++, Java or Verilog. Program 1 is a very simple snippet of RDL which shows the basic structure of an RDL file, minus the usual copyright and comment headers [46].

Program 1 consists of four namespaces called "Foo", "Base," "NonLocal," and "UseRename" inside of the base design namespace which in RDL may be referenced as `::`. In each of the namespaces declared in this file there appears a single message type declaration (see Section 5.2): "DWORD," "BIT," and "LOCALBIT." The `include` declaration at the top of the file will include the contents of the file "Foo.rdl" as if they had been declared inside the namespace "Foo" directly in this file, and will be discussed in Section 5.1.4.

Because RDL is a netlisting language, without a behavioral component, every line of code (terminated with a `;`) is a declaration of existence for some construct. Static declarations, as all of those in Program 1 are, can be viewed as ways of giving a name (a static identifier) to something. Dynamic declarations, which will be discussed elsewhere (see Section 6), mainly state that one object is constructed by assembling others.

In order to keep RDL readable, all declarations are of the general form `keyword value name;`. This is done to match the syntax of the most widely used languages. Though a `name:value;` format would reduce the verbosity the decision was made arbitrarily and is under review (see Section 16.4). It should be noted that the value may range from a message type, to a complete namespace as shown in Program 1, to a parameterized expression which evaluates to either of these.

Finally, the declarations of the types `::Base::BIT` and `::UseRename::LOCALBIT` deserve more explanation, which is deferred to Section 5.1.5.

Program 1 Basic RDL File

```
1 include "Foo.rdl" as Foo;
2
3 namespace {
4   message bit<0x20>    DWORD;
5 }                      Base;
6 namespace {
7   message bit<1>       ::Base::BIT;
8 }                      NonLocal;
9 namespace {
10  message ::Base::BIT  LOCALBIT;
11 }                     UseRename;
```

While it is often considered good form to declare all entities (message types or units for example) giving them individual names, RDL does not require this. Thus the name something is declared to have, and its actual value are interchangeable are the lexical level. This means that `message ::Base::bit LOCALBIT` and `message bit<1> LOCALBIT` accomplish the same thing.

5.1.4 Namespaces

Namespaces are the RDL solution to library isolation and management in a collaborative environment, a problem which has plagued the older more stable HDLs. Declarations in one namespace may reference those from another (using compound static identifiers as on line 10 of Program 1), but they are, by default entirely isolated. A namespace in RDL, as in other languages, significantly increases the modularity of the language by allowing researchers in one group to create their design independently of another group and still share code later. Of course namespace names are entirely at the discretion of the RDL author, subject to basic character set requirements (*e.g.* no spaces), and are irrelevant to the language and compiler.

Because namespaces are hierarchical and arbitrarily named, like directories in a file system, a scheme similar to the Java tradition of using DNS

names in reverse order might be appropriate as a coding style. On the other hand, the small number of RDL libraries written to date, particularly the lack of a "standard library" has left this a somewhat open issue and a matter of taste.

Because namespaces names are specified externally by `include` statements, the end designer of a system is free to build their own hierarchy by "mounting" declarations at will. The inherent namespace declaration from an `include` statement ensures strong isolation of namespaces between projects and libraries, while the ability to keep multiple declarations in one file reduces the overhead of maintaining a library which consists of many small RDL declarations.

Above we suggested several coding standards, but no consensus has been reached about the proper style for name isolation in practice.

Though we have compared namespaces to standard object oriented languages, they are not objects. Namespaces represent scopes, in the formal language sense, and provide no mechanism for inheritance or importing one namespace in to another, as include creates a child namespace. The hierarchy provided by namespaces has no meaning at the semantic level, and is purely for human consumption.

This example also shows the use of qualified identifiers, whereby the coder has used a declaration in `namespace UseRename` to give `::Base::BIT` a local name `LOCALBIT`. The details of this statement deserve a minor explanation: a qualified static identifier is very similar to a file path, navigating through namespaces rather than folders. The `::` qualified in static identifiers is exactly like the `/` qualifier in UNIX path names, both for specifying the root, and relative path segments. The only difference, is that rather than writing `../../` to specify the grandparent namespace, one would write `::2::`, a shorter and less ambiguous notation than the alternative `::::`.

As a final note, RDL uses late binding which allows declarations to appear in any order without regard to their use. Because RDL is not a sequential language, it would make no sense for things to be declared "before" their use anyway, except to make life easier for a lazy compiler writer.

5.1.5 Non-Local Declarations

RDL allows all constructs, in Program 1 messages are shown, to be declared in one namespace to belong to another, in order to provide high level algorithmic parameterization. This allows a designer to use the declaration of the type `message BIT` in the `namespace Base` without regard to the fact that the declaration for `Base` includes no such type. The coder has created the `namespace Base` and used a declaration in the `namespace NonLocal` to extend `Base`. In effect `::Base::BIT` is a kind of parameter, whose value is determined by the inclusion or exclusion of the `namespace NonLocal` in the project.

To make matters crystal clear, the two snippets of code in Program 2 are treated as identical in RDL. The only difference is that though two namespaces are shown as being declared in the same file, they could in fact be completely separate and been developed independently.

Program 2 Non-Local Declarations

(a) Non-Local Declaration

```
namespace Base {
  message bit<0x20> DWORD;
};
namespace NonLocal {
  message bit<1> ::Base::BIT;
};
```

(b) Local Declaration

```
namespace Base {
  message bit<0x20> DWORD;
  message bit<1> BIT;
};
namespace NonLocal {
};
```

With the addition of the `include ''Filename.rdl'' as NewNamespace` statement, the ability to make non-local declarations, such as `BIT` in this example, becomes a powerful mechanism for independent RDL development. A developer may create a namespace, and reference objects within it which are not actually declared there. Independently another developer may add declarations to this namespace using non-local bindings. The combination of non-local declarations and late binding allows for a kind of algorithmic parameterization, whereby a declared unit can instantiate a unit which is not declared until later, by another researcher in another file.

5.1.6 Parameters

Parameterization is particularly vital for building any reasonable library of unit implementations as without parameterization, any such library would either be extremely verbose or laughably limited. For example Program 3 shows two units, one a source of generic data and the other a sink, which are useful in many code examples and would be impossible to specify without parameters.

Program 3 Dummy.rdl

```
unit <Width> {
  output bit<$Width> Out;
} DataSource;

unit <Width> {
  input bit<$Width> In;
} DataSink;
```

Though non-local declarations (see Section 5.1.5) serve the purpose of algorithmic parameterization, they lack the elegance and clarity, both for humans and automated tools, of standard parameterization. In particular, they do not allow for differing parameter values for the various instances of a single unit, platform or type. This led to the introduction of a more standard parameterization mechanism for RDL2, which is modeled on C++ templates and syntactically similar to Java generics.

Parameters in RDL2 can take the place of strings, numbers and static identifiers. This means that, for example, array declarations may have parameterized size and channels may have parameterized timing models. Parameters can also provide a per-instance form of algorithmic parameterization by passing a static identifier to a unit, which will then instantiate the unit named by that parameter.

In the context of declarations of the `keyword value name;` format, parameterization turns the general syntax to `keyword <formals> expression <actuals> name;`. The formal parameter list is a comma separated list of parameters names, dynamic identifiers, possibly with default values, for example `<foo = 7>`. The actual parameter list of course must specify a value for the parameters, and the expression then must evaluate to the proper type (*e.g.* a message if the keyword is "message") when parameter substitution is complete.

Particularly important is that parameter names, when referenced must be prefixed with a `$` to clearly separate them from dynamic and static identifiers. Interestingly, parameters may be referenced non-locally *e.g.* allowing a unit declaration to use the value of a parameter on one of its children by naming it using a dynamic identifier. This allows for a kind of reverse parameterization, where parameters propagate from lower to higher levels of the dynamic (instance) hierarchy.

Note that there is no requirement that all parameters be given a value, particularly because parameters on units are passed unmolested to the unit implementation in the implementation language (see Section 9.5). This has the consequence that an unassigned, unused parameter cannot be caught by any RDL tools but instead must be caught by whatever tools process the unit implementations (FPGA synthesis or a compiler), something which may need to change in the future. In particular, we worry about the consequences of this in comparison to the "feature" of Verilog which allows the use of undeclared wires, making it very hard to debug designs with simple typos.

5.1.7 Inference

In addition to simple declarative parameterization, RDL includes parameter inference, meaning that while a parameters value may be explicitly set in a declaration or instantiation (see Section 6.1), it may also be inferred from use. The unification algorithm, which both infers values for unspecified parameters and checks for errors, is simple, and though capable of unifying identical values, will always fail should two different values for a parameter be inferred.

As an example of parameter inference Program 4 and Figure 18 show an adder unit where the number of inputs and their bitwidth are parameterized. Inference allows the user of this unit to specify the width of no more than a single port. As all of the ports are specified to have the same width, RDLC2 will infer the width of other ports from a single one.

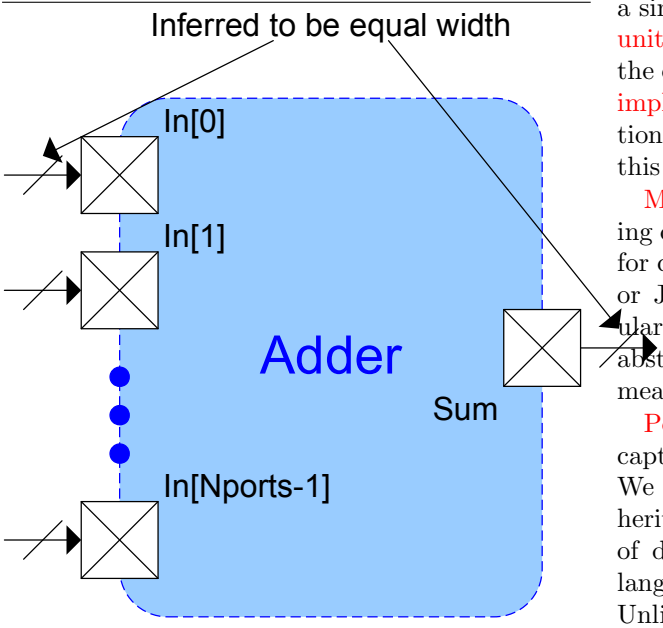
Program 4 Adder.rdl

```
unit <Width, NPorts> {
 input bit<$Width>[$NPorts] In;
 output bit<$Width> Sum;
} Adder;

unit {
  channel InChannels[2], OutChannel;

  instance Dummy::DataSource<32>
    Source1(InChannels[0]),
    Source2(InChannels[1]);

  instance Adder
    Adder(
      [index#0](InChannels[index#0]),
      OutChannel);

  instance Dummy::DataSink
    Sink(OutChannel);
} AdderExample;
```

For a complex example of parameter inference including how it interacts with multidimensional arrays, please see Section 7.1

As a direct consequence of parameter inference, we have omitted operators over parameters from

Figure 18 Adder



the RDL2 specification. In particular, though simple arithmetic operators might not pose a problem, the problem of specifying a parameter unification algorithm under generic operators was not one we wished to address. Common requests include examples such as $OutputWidth = log_2(NPorts) + Width$, but RDLC2 does not support expressions involving parameters, and RDLC1 does not support parameters at all. This decision proved invaluable in the short term, though it is likely to change in our future work (see Section 16.4.1).

5.2 Messages, Ports & Terminals

The RDL type system exists to enable the uniform and clear communication of unit interfaces among both researchers and automated tools. Having introduced the basic lexical structure and syntax of RDL in Section 5.1, we procede in this section to describe the three type hierarchies of RDL: messages, ports and terminals. Note that this section does not explain theoretical model behind these concepts (see Sections 3 and 4).

While messages and ports are obviously related in the sense that ports are the connection points between units and channels, terminals are included in this section as these are the three constructs in RDL which have typing rules.[1] Units technically have type information associated with them, but in a simplistic way, as we never attempt to determine unit equivalence. This is technically a drawback of the current language specification as it hinders unit implementation sharing during mapping (see Section 8.4) and implementation multithreading, but this is well beyond the scope of this section .

Message types are intended to capture the meaning of message fields for marshaling, debugging and for communication to languages like SystemVerilog or Java which support these features. In particular, structures, arrays and unions help raise the abstraction of messages from simple bit-vectors to meaningful information.

Port structures are the mechanism whereby RDL captures interface definition and implementation. We use port structures instead of some form of inheritance, which is difficult to codify and use, and of dubious value as RDL, unlike object oriented languages, does not capture unit implementations. Unlike messages which are obviously visible to the unit implementations, port types exist primarily at the language level, and include structures, unions and arrays of ports which can be easily connected to each other. Not only is this powerful syntactic sugar for creating individual channels, but it allows these complex port types to act as a flexible interface abstraction.

Terminal structures are analogous to port structures. However, where port structures allow the target designer to specify complex unit interfaces, terminal structures allow the host designer to specify complex platform interfaces. Complex terminals provide all the same benefits as complex ports, though of course multiple terminals (more than two) may be connected by one link.

In the following sections we will discuss first the base types (see Section 5.3) of messages, ports and terminals. We will also cover the modifiers which may be applied to ports (see Section 5.4).

5.3 Base Types

In this section we cover the various type operators available for constructing complex message, port and terminal types. We will draw example code from Program 5 which is included as an example in the standard RDLC2 distribution. Note that these examples, and many like them are also the basis of the RDLC test suite (see Section 9.6).

This section discusses the available types in the context of type declarations, which allow the types to be given names (static identifiers), whereas usage of these types will be explained in elsewhere

[1] We have produced formal typing files, not reproduced here.

Program 5 DataTypes.rdl

```
message event MessageEvent;
message bit<8> MessageSimple;
message bit<8>(3) MessageArray;
message munion {
  event FieldA<1>;
  bit<8> FieldB<2>, FieldC<20>;
} MessageUnion1;
message munion {
  bit<7>(2) FieldA;
  bit<2> FieldB, FieldC;
} MessageUnion2;
message mstruct {
  bit<27> Address;
  bit<256> Data;
} MessageStruct;

port MessageStruct[10] PortArray;
port punion {
  covariant event Field1<2>;
  contravariant bit<2> Field2<3>;
} PortUnionTagged;
port <awidth = 27, dwidth = 256>
    pstruct {
  bit<$awidth> Address;
  bit<$dwidth> Data;
} PortStruct;
```

(see Section 6). Type declarations in RDL provide more than a simple short name for a type, they actually create a distinct type. For example the statement `message foo bar;` will create a message type `bar` which though it has the same underlying representation as `foo` and is in many ways indistinguishable, is not equal. This is a consequence of the hardware focus of the target model, wherein it is important to distinguish between, for example, 8-bit quantities which represent integers, and those which represent ASCII characters.

5.3.1 Events & Bits

The base type for messages is a simple bit-vector, with a non-negative length as shown in Program 6. This was chosen as being a good match both for detailed architectural specifications with which a RAMP researcher might want to experiment and the underlying FPGA platforms. Furthermore, bit-vectors are easily portable across both hardware and software platforms and, when used with proper abstractions, will be efficient in both domains.

While the benefit of sending messages with positive lengths should be obvious, this allows a target design to move data about, it is interesting to note that 0-bit messages like `MessageEvent` are also useful. In particular these could be used to carry temporal information like interrupts or synchronization messages such as for a distributed barrier.

Program 6 Simple Messages

```
message event MessageEvent;
message bit<8> MessageSimple;
```

As a side note, the RDL keyword `event` is equivalent to `bit<0>`, according to the RDL type system. This allows bit-vector lengths to be parameterized, where the parameter can be 0. The `event` keyword in simply syntactic suger.

The choice of `<>`, the parameter list markers, to denote bit-vector lengths in RDL2 is a result of the need to unambiguously separate these from message, port and terminal arrays as described in Section 5.3.3. This was a change from RDL1, which used square brackets for bit-vector length partly because it did not have array support.

5.3.2 Terminals & Ports

Unlike messages whose base-types are bit-vectors, ports and terminals have a more abstract notion of base type, as shown in programs 7 and 8. For ports the base type is simply the type of messages sent through that port. As shown in Program 7, this means that ports may be declared both in terms of previous declared message types or the base message types listed above.

Program 7 Simple Ports

```
port event PortEvent;
port MessageSimple PortEvent;
```

However for terminals, the base type is an opaque invocation of an RDLC2 plugin. Plugins (see Section 6.5) are acceptable because terminal types need not be particularly analyzable by the tools, and because the wide array of possible links (see Section 4.3) would make this prohibitive anyway. Using plugins therefore allows a large amount of freedom in the range of technologies and implementations for links (see Section 10.4.1).

Program 8 Simple Terminal

```
terminal ("TestLink") TestTerminal;
```

5.3.3 Arrays

Given that RAMP is dedicated to producing simulations of multicore architectures, its quite natural that RAMP designs should scale arbitrarily and easily. Having completed RDL1 as an exercise in implementing the target model, RDL2 it was realized would need much better parameterization and scaling support. As the simplest form of scaling, linear arrays of ports, messages and terminals allow RDL descriptions to be significantly more terse, and general at the same time, a clear win.

Of course spatial locality demands that such architectures scale in two dimensions, or even three, meaning that RDL support arbitrary n-dimensional arrays of types. Any such limitation on the dimensionality of types would have been artificial, and serve no purpose other than to ease the life of a lazy compiler writer. Further, the authors have been, in the past, frustrated by similarly lacking tool support in for example various Verilog synthesis tools.

Program 9 Array Types

```
1 message bit<8>(3) MessageArray;
2 port MessageStruct[10] PortArray;
3 terminal TestTerminal[5] TerminalArray;
```

Note that the array bounds for messages are specified using `()` whereas all other array bounds in RDL are specified using the more conventional `[]`. This is done to disambiguate message arrays from port arrays in simple cases like `port bit<2>(4)ASimplePort` which is a simple port carrying an array message from `port bit<2>[4] AnArrayPort` which denotes an array of ports each carrying messages. This is necessary to avoid forcing the RDL writer to pre-declared names for all message types. This is important because as stated above type declarations technically create new types which are distinct from the ones on which they were based, making such pre-declarations not only verbose but contrary to the goal of highly interoperable, separately designed units. Of course such pre-declarations, which effectively force abstract data types are still good stylisticly, but we have no wish to require them at the language level.

Lest there be any confusion on this point, a message, no matter how complex its structure is always atomic at the target level and carried on a single channel implementation. By contrast a connection between two complex ports is syntactic sugar for writing out the complete port and connection descriptions by hand, the individual, unstructured ports each receive their own independent channel implementations and separate timing models.

5.3.4 Structures

Structuring of messages, though obviously not required for unit implementation given that Verilog is a host language, allows RDL to capture the semantics of the messages it touches. Given the goal of allowing units developed independently to interoperate, the ability to specify a clean interface in terms of structured ports carrying structured messages was deemed vital to the success of RDL early on and is supported, although with slightly different syntax by RDL1 and RDL2.

Program 10 Struct Types

```
1 message mstruct {
2   bit<27> Address;
3   bit<256> Data;
4 } MessageStruct;
5 port pstruct {
6   bit<27>  Address;
7   bit<256>  Data;
8 } PortStruct;
9 terminal tstruct {
10   TestTerminal Test;
11   RS232Terminal Serial;
12 } TerminalStruct;
```

Shown in Program 10 are examples of message, port and terminal structures. Message structures are mostly opaque to RDLC, being useful mostly to marshaling and debugging code.

Port and terminal structures, however, allow complex unit and platform interfaces to be connected easily, by simply connecting two port structures of identical type together for example. While terminals inherently are bidirectional, ports have direction, and thus port structures may contain ports which go either with or against the primary direction of the port structure. For more information, see Section 5.4.4 below.

Again, messages, no matter how complex their structure are atomic and transmitted over a single channel, whereas port and terminal structures are simply syntactic sugar for declaring multiple ports and terminals with their own separate channels and links respectively.

5.3.5 Unions

Message unions, as shown in Program 11, first appeared in RDL1 as a response to the simple example of a DRAM request message, which will sometimes contain write data and sometimes simply an address to read. Unions in RDL allow the unit implementor to send different kinds of messages between units at different times without the overhead

of multiple channels or simply sending the largest message.

In addition to providing structure for any RDL debugging tools, and a tighter interface definition language to collaborating researchers, message unions also allow a link to exploit the channels average bandwidth for higher performance, rather than being forced to provide the maximum bandwidth at all times. By marshaling (see Section 4.2.1) union messages to avoid transmitting the unnecessary bits, a low performance link can easily emulate a high performance, though often unused, channel. Exploiting this difference between average and worst cases, requires that the tools be aware of the difference, hence the addition of message unions to RDL.

Program 11 Union Messages

```
message munion {
  event FieldA<1>;
  bit<8> FieldB<2>, FieldC<20>;
} MessageUnion1;
message munion {
  bit<7>(2) FieldA;
  bit<2> FieldB, FieldC;
} MessageUnion2;
```

messages unions in RDL are tagged, primary so that unit implementations and the RDL tools share a common mechanism for distinguishing which field of a union is active in a given message. Not only does this allow interface specifications to be clear and unambiguous, it also means that RDL-centric tools can be aware of the meaning of unions. Finally, un-tagged unions make no sense as there is no way to recover the meaning of the data later. This means that C programmers, for example, are simply required to tag their unions using external means which the tools are not aware of. Note that by default RDL will assign tags from 0 sequentially to the fields list in a union for consistency, but this may be overridden as shown in `MessageUnion1` of Program 11.

Program 12 Union Ports

```
port punion {
  event Field1<2>;
  bit<2> Field2<3>;
} PortUnionTagged;
port tunion {
  TestTerminal FieldAlpha<75>;
  RS232Terminal FieldBeta<6>;
} PortUnionTagged;
```

Port and terminal unions, shown in Program 12, are unlike port and terminal structures and arrays in that they provide more than syntactic sugar. Unions provide a way for a unit or platform designer to stipulate that one and only one of the sub-ports or sub-terminals may be connected at a time. This is particularly useful for platform interfaces which are mutually exclusive and generalized unit implementations which can accept inputs in a variety of formats.

As an example, consider a DRAM input port union as shown in Program 13, which can accept write data with the commands using the `DRAMInput.Together` port, or separately using the `DRAMInput.Separate` ports. Of course it is the responsibility of RDLC to pass simple information, like which port out of a union is actually connected, to a unit implementation by specifying the tag for the connected port or terminal as a constant.

Program 13 DRAM Input Union

```
port punion {
  DRAMCommandMessage Together;
  port pstruct {
    DRAMCommand Command;
    DRAMData WriteData;
  } Separate;
} DRAMInput;
```

Again, messages, no matter how complex their structure are atomic and transmitted over a single channel, whereas port and terminal structures are for declaring multiple ports and terminals with their own separate channels and links respectively.

5.3.6 Summary

The RDL type system exists to enable the uniform and clear communication of unit interfaces among both researchers and automated tools. In this section we have shown code snippets for the base message, port and terminal types and the modifiers while explaining their use. We have also given the motivation for the existence of these types and their details.

Aside from the base types, we have also covered the type structures and type equivalence of RDL which requires that declaration of a type, creates a separate new type not equivalent to what it is based on. This allows RDL to associate meaning with types beyond the width and structure of them, a necessary condition for effective interface specifications.

5.4 Modifiers

This section covers the modifiers which may affect a message, port or terminal type. All of these have been added in direct response to either initial test cases, or particular applications (see Sections 13, 14 and 15). As such the list of modifiers is only like to grow, and some form of unified mechanism for annotation is likely to be appropriate in the future (see Section 16.4).

In the context of the standard declaration syntax, `keyword value name`, modifiers appear after the key word, as in `keyword modifiers value name`. If there are multiple modifiers they should appear in a space separated list, and needn't be in any particular order.

5.4.1 Alias

While declaring a new type based on an old one normally creates a completely unrelated new type, this keyword can be added to change this. During a large design it may be necessary for some types to be given shorter names. The alias keyword enables this by stating that the newly declared type is actually only an alias for the original type, rather than a separate type. For example in Program 14 the types `A` and `B` are equivalent, where as `C` despite being based on `A` is equivalent to neither `A` nor `B`.

Program 14 Type Aliasing

```
1 message bit<2> A;
2 message alias A B;
3 message       A C;
```

The alias modifier can be applied to any type declaration.

5.4.2 Optional

On the one hand it is a serious error for a port to remain unconnected, and the RDL tools should be capable of detecting this. On the other hand it is important for the purposes of writing general units for the use of some ports to be optional. Thus declaring a port, either within a unit (see Section 6.2) or in a port structure to be optional allows the instantiation of the unit without all its ports being connected.

The optional keyword can be combined with port structures to interesting effect. For example a union port which is optional requires that 0 or 1 of its sub-ports must be connected, whereas an optional structured port requires that 0 or all of its sub-ports be connected.

As a side note, as with port unions, RDLC will inform unit implementations as to which ports are connected. For preference this information should be provided for all ports for uniformity rather than just for the optional ports.

The optional modifier applies only to ports. Terminals are always optional, as there is no requirement that any pair of platforms be connected at all unless this is required by the particular implementation mapping. messages are always optional by virtue of the elastic nature of channels, a decision which may be re-evaluated in the future (see Section 16.4) by the addition of more channel models.

5.4.3 Direction

RDL is agnostic about the direction of terminals, as it has no particular use for this information, meaning that they may in fact be bidirectional. messages of course must travel along the channel in the prescribed direction. However, the direction of channels is determined by the direction of the ports to which they are connected.

The input and output modifiers are used, not in type declarations but in unit declarations to specify the directionality of ports. Note that while a complex port may be an input or output, its fields may vary from this as described in Section 5.4.4 below.

5.4.4 Variance

As port structures serve the purpose interfaces in RDL, it is critical that the language be able to specify bidirectional port groupings. In particular, consider the example (see Section 7.5) of a CPU whose interface to memory must include a way to move data in both directions for read and write.

Program 15 shows how one might specify a generic memory interface for both loads and stores as a composite of a command & write port with a read port. This port structure could then be used as an input on the memory, and an output on the CPU. Because the `MemIO.MemIn` port is `covariant`, the channel connected to that port will have the same direction as the `MemIO` port. Because the `MemIO.MemOut` port is `contravariant`, the channel connected to that port will have the opposite direction as the `MemIO` port.

Covariance and contravariance are relative to the parent port not to the overall port, meaning that port structures can be easily nested.

The user may omit the modifier `covariant`, as it is obviously the default. However it is considered good style, if the `contravariant` modifier is in use in a particular port structure to explicitly label all the `covariant` ports as such.

Program 15 Port Variance

```
port pstruct {
  covariant message munion {
    LoadRequest       Load;
    Store             Store;
  }                   MemIn;

  contravariant message
    LoadReply         MemOut;
}                     MemIO;
```

5.4.5 Opaque

In order to build a reasonable standard library of units, it is desirable to sometimes create a unit which does not operate on the data it processes. In particular, imagine a memory structure of some kind such as a FIFO (see Section 7.3) or RAM which merely stores data for later retrieval. It may be desirable for this unit to accept messages of any type without regard to their contents.

In order to accomplish this, the `opaque` keyword should prevent any RDL tool from generating message formatting logic, in particular packing (see Section 4.2.2) and marshaling (see Section 4.2.1) logic for a particular port. In combination with a unit level parameter for the type of the port, this will allow the unit to accept or send messages without tying the unit implementation to a particular message type.

5.5 Conclusion

RDL was designed both to help formalize RDF, and to capture systems within them for manipulation by automated tools, in particular the RDL Compiler (RDLC) and is a declarative system level language, and contains no behavioral specification for units, relying on existing design libraries and languages. In this section we have presented the lexical structure, syntax and semantics of the RAMP Description Language (RDL) particularly as they relate to the goals of RAMP (see Section 2.2). In particular the type system (see Section 5.2) is vital to the community building efforts of RAMP, as it allows researchers working with disparate HDLs or software languages to define common, easy to understand interfaces. Section 6 covers the netlisting aspects of RDL, through which target and host systems are constructed.

The complete RDLC1 and RDLC2 code, examples and documentation downloads can be down on the RAMP Website [9]. In addition to the complete source code, the downloads include instructions for compiling and mapping to a Xilinx FPGA, an Altera FPGA or a Java platform and a copy of this thesis.

Chapter 6

RDL Dynamics

RDL provides an abstraction of the locality and timing of communications, enabling timing accurate simulation and cross-platform designs. In addition, various tools (see Sections 11 and 12) will add distributed system level debugging (see Section 16.4), and perhaps even power estimation tools.

In order to build useful simulations it is imperative that we not rely on implementing the system we wish to study, but provide some way to model it. Furthermore, any such simulation must obviously scale beyond the confines of a single FPGA. Automating the virtualization of time and cross-platform support requires some tool to examine a system at the structural level, rather than the RTL level of typical HDLs like Verilog or VHDL. RDL therefore provides a high level description of the system being simulated, the system performing the simulation and the correspondence between them.

For both units and platforms, RDL is a combination of a structural modeling language and a simple netlisting language, such as a subset of Verilog or VHDL might provide. This section covers the netlisting aspects of RDL for both target (see Section 3) and host (see Section 4) systems, in other words the "at run time" or dynamic systems.

At the current time RDL includes support for hierarchical namespaces (see Section 5.1), messages & ports (simple, structured, union and arrays) (see Section 5.2), units & platforms (leaf, hierarchical and arrays) (see Section 6) and mappings from *e.g.* units to platforms (see Section 6.4). RDL does not, and will never, provide for the specification of leaf unit behavior, as it is aimed at tying together existing designs, and there are enough behavioral languages in common use. Such a language might be useful, and could easily be integrated with RDL, given the formal specification of the syntax, but the resulting language would not be RDL.

Unit designers must produce the gateware (Verilog for RDLC1 & 2) or software code (Java for RDLC1) for each unit in their chosen language, and specify, in RDL, the types of messages that each input or output port can carry as well as the structure of the target and host systems. For each supported language, RDLC automatically generates a wrapper which interfaces from the unit implementation to the links implementing the channels and provides target cycle firing control (see Section 4.2.4), if a simulation is to be built. In addition the links implementing the various channels are generated automatically, using an extensible framework architecture (see Section 10), from the connections and hierarchy specified in the RDL source. Of course all of this takes place under a set of mappings which allow a target design to be split between many platforms.

The biggest cost associated with using RDL is the time taken to describe a design using target model and RDL. Second, is the area, time and power to implement this model, all of which are easily controlled and are functions of the simulation rather than the language. RDL provides no free lunch here, and is designed to be simple and transparent rather than providing a complex abstraction. In effect this means that as a designer "you get, and must pay for, what you ask of RDL." This is a major benefit of RDL above other system level languages, as it does nothing to obstruct the skilled hardware designer.

The biggest benefits of RDL are: deterministic timing, clean interface encapsulation, implementation language independence and the sharing of units between researchers that these things allow.

6.1 Netlists

The main goal of RDL is to clearly describe both parameterized target (see Section 3) and host (see Section 4) systems, allowing one to succinctly map between them. Section 5 discussed the basic lexical structure, syntax and the type system of RDL. In this section we lay out the syntax and semantics for the declaration of units & platforms and channels & links, including the assembly of these

to describe complete systems. Collectively we refer to these elements as "dynamics" because they form the dynamic, running system, as opposed to the RDL namespaces and types which are constructs of the language and tools.

Many of the examples in this section are drawn from the RDL "CounterExample" (see Section 7.4), which is a complete RDL description of a very simple system built around a counter.

6.2 Units & Platforms

A complete system at either level is represented in RDL as a hierarchical netlist of the appropriate building blocks. The cornerstones of the target and host models respectively are units and platforms.

Netlist representations are an ideal match to genuine implementations of target systems making them highly suitable for a certain class of simulator: structural models. While some sub-projects within RAMP eschew these simulators for more abstraction behavioral models, we believe that all models must be structural, even if only at the highest level. Capturing even these high level system descriptions in RDL allows them to be parameterized and automatically implemented. Thus while we focus heavily on structural models, we believe RDL applies just as well to the highest level of so-called behavioral models. This generality is yet another reason RDL does not include the behavioral specification for units.

As RDL is a hierarchical netlisting language without behavioral specifications, both unit and platform declarations establish the existence of said objects, as well as their interfaces. Thus there is a distinction between a leaf level unit which must be implemented in another language (Verilog or Java for example) and hierarchically constructed units which exist only within RDL as a convenient abstraction. Only leaf level platforms represent places where unit implementations can be run, whereas hierarchically constructed platforms typically represent physical (PCB) or administrative (PC cluster) domains.

The advantage of this simple view is that the hierarchy may be ignored by some RDL processing tools (see Section 8.4) if convenient, a trick particular applicable to target systems. For example an implementation in software may schedule unit execution (see Section 4.4.2) on a single processor without regard to the structure of the target system. The disadvantage is that things like debugging or board-level concerns (*e.g.* power or temperature) are difficult to capture. To this end while the primary RDL constructs treat the leaves as important and ignore the hierarchy, RDL plugins (see Section 6.5), such as those for power and temperature estimation, will often take a different approach.

6.2.1 Declaration

The first and most important things one can do with a unit or platform is declare it. As shown in Program 16, leaf level unit and platform declarations include a name, a formal parameter list, optionally with default values, and the port or terminal list. For units this will specify directions for all of the ports. For platforms this will also include specifying the language in which any wrappers or links mapped to this platform should be generated.

Program 16 Simple Unit & Platform

```
1 unit <width, saturate = 1> {
2   input bit<1> UpDown;
3   output bit<$width> Count;
4 } Counter;
5
6 platform {
7   language "Verilog";
8   terminal ("TestLink") TestTerminal;
9 } Tester;
```

While RDL parameters allows for parameterized system descriptions, they are also intended to allow parameterized unit implementations. Parameters on leaf units and platforms, like the `saturate` parameter to the `Counter` unit in Program 16, should be passed to the actual unit implementation or platform in question after mapping. In this particular example the `saturate` parameter is designed to select whether the counter will saturate at its upper limit or merely roll over.

It should be noted that declarations will typically include some form of plugin invocation, both for units (see Section 10.3.1) and platforms (see Section 10.4.2) which are special. This conflation of plugins with base RDL syntax allows a significant flexibility in the language, while simplifying its syntax, but is likely to be changed in future specifications to avoid confusion.

6.2.2 Instantiation

In order to build larger systems it is desirable to construct them hierarchically, thereby abstracting the implementation details to a more manageable form. In RDL this means constructing hierarchical units and platforms by instantiating leaf level units and platforms, as shown in programs 17 and 18.

Of course units may only be instantiated within units and platforms within platforms. Target and

host systems are kept separate in order to ensure that at least the RDL descriptions of targets are entirely portable, and that hosts are reusable. The process of creating an implementation of a target system for experimentation is called mapping (see Section 6.4) and is orthogonal to instantiation.

Program 17 Hierarchical Unit

```
unit <width> {
  instance IO::BooleanInput
      BooleanInputX;
  instance Counter<$width> CounterX;
  instance IO::DisplayNum<$width>
      DisplayNumX;
} CounterExample;
```

Program 17 shows an example of a complete target system, constructed hierarchically from three lower level units (including the counter from Program 16). This is considered a complete system because the `CounterExample` unit has no ports for I/O. Note that this does not mean that the system does not interact with the outside world, merely that this abstraction has been encapsulated in the `IO::BooleanInput` and `IO::DisplayNum` units (see Section 4.4.3), as one might guess from their declaration in the `IO` namespace. A unit with ports might still be a complete target system so long as those ports are optional. A unit without ports should never be part of a higher level target system, as it would have no way of interacting with the rest of the simulation, making it quite pointless.

Note that the `CounterExample` unit is allowed to have an unbound parameter, `width`, which enables the specification of parameterized system descriptions. In this case it as simple as the width of the counter being simulated, but more realistic examples of RAMP systems might include parameters for the number of processor cores or even the network topology.

Program 18 Hierarchical Platform

```
platform {
  instance CaLinx2 Board0, Board1;
} DualCaLinx2;
```

RDL includes support for hierarchical platforms, because the RAMP project seeks to simulate targets at the scale of thousands of processors, and there are no individual FPGAs or single processor computers capable of performing such a simulation in a reasonable amount of time. Program 18 is an example of a platform consisting of two CaLinx2 FPGA boards [33].

Whether creating hierarchical platforms or units the basic form of an instantiation is of the form `instance \gls{unit} instancename, instancename;`, allowing one to create multiple identical instances in one statement. The instance name is a dynamic identifier which can be used to identify a unit or platform within an instantiation hierarchy, for example `DualCaLinx2.Board0` denotes the first board in Program 18.

Of course the examples in this section are missing some key components, particularly there are no channels or links, meaning that these examples are quite useless. We will correct this in Section 6.3 below.

One of the interesting things about RDL is that because a type or unit may be declared as it is instantiated, it has access to all the parameters of an enclosing scope. This means that one can write code such as that shown in Program 19, because the `Board0` and `Board1` platforms are declared as they are instantiated within the scope of the `language` parameter.

Program 19 Parameter Scoping

```
platform <language> {
  instance {
        language $language
  } Board0, Board1;
} ParameterizedLanguage;
```

6.2.3 Arrays

Aside from implementation parameters like `width` in Program 17 above, RDL2 allows the creation of models whose structure is parameterized. In particular RDL2 allows for arrays of units or platforms to be declared, as shown in Program 20. Though RDL2 does not include Turing-complete parameterization, it has no parameterizable if, for or state constructs, this can be achieved with plugins (see Section 6.5).

Program 20 Platform Array

```
platform <clustersize = 2> {
  instance CaLinx2 Board[$clustersize];
} ClusterCaLinx2;
```

Program 20 is almost entirely equivalent to Program 18, except that the number of CaLinx2 boards [33] in the cluster has been parameterized. It is also worth noting that the array bounds for units and platforms are enclosed in `[]` the same as array

bounds for ports and terminals. The array bounds may also be omitted, leaving just `[]` when RDL parameter inference is expect to infer the size of the array (see Section 6.3.3).

Though programs 20 only shows a one dimensional array, RDL supports arrays of arbitrary dimensionality. Note that the dimensionality of an array is statically declared however, meaning that one cannot *e.g.* use a parameter to decide the dimensionality of a netlist, an action which might be useful in network simulations.

Again, note that this example includes no links, making it quite artificial, a shortcoming we will correct in Section 6.3.3, where we will also discuss the interaction with port and terminal arrays.

6.3 Channels & Links

With the addition of links and channels to connect them leaf level units and platforms can truly be assembled to create higher level, abstract units and platforms. In this section we outline the syntax for declaring channels and links, and connecting them to units and platforms respectively.

6.3.1 Instantiations

Channels may only be instantiated within hierarchical units and links within hierarchical platforms. Program 21 shows some simple example instantiations of two channels and a link. Like unit and platform instantiations, similar channels or links may be declared on one line using a comma separated list of their names.

Program 21 Simple Link & Channel

```
1 unit {
2   channel InChannel, OutChannel;
3 } CounterExample;
4
5 platform {
6   link UART;
7 }  DualCaLinx2;
```

RDL2 supports the specification of a channel timing model after the keyword `channel` and before a list of instance names, all of which will have the same timing model. RDL2 includes two major timing models, one called `fifopipe` which supports the full timing model including all four timing parameters (see Section 3.3) and another called `pipe` which has only a forward latency, and models an inelastic pipeline. Though the language includes these keywords, their proper implementation is dependent upon the link to which the channels are mapped, and the code which generates that link (see Section 4.3).

6.3.2 Connections

Channels and links aren't particularly useful unless they are connected to ports and terminals respectively. RDL, in an effort to be flexible, supports three methods of connecting both channels and links, all of which are shown in Program 22.

Program 22 Connected Counter

```
1 unit <width> {
2   instance IO::BooleanInput
        BooleanInputX(Value(InChannel));
3   instance Counter<$width> CounterX(
        InChannel, OutChannel);
4   instance IO::DisplayNum<$width>
        DisplayNumX;
5
6   channel InChannel, OutChannel { ->
        DisplayNumX.Value };
7 } CounterExample;
```

First, on line 2 of Program 22 the channel `InChannel` is connected to the `Value` port of the unit instance `BooleanInputX`. Note that this style of connection mirrors Verilog connections, with the subtle difference that there is no . before the port name `Value`. Connections in this format occur in within parentheses after the instances name, and consist of a comma separated list of connections, each one of the form `PortName(ChannelName)`. These are referred to as "named connections" because the port being connected is named.

Second, on line 3 of Program 22 the channels `InChannel` and `OutChannel` are connected to the first and second ports of the instance `CounterX` respectively. Again, this format derives from standard Verilog connections, differing from the above format only in that the elements of the comma separated list are of the form `ChannelName`, and are matched to ports by the order in which the connections, and of course the ports, are declared. These are referred to as "positional connections" because the port being connected is determined by the position of the channel within the connection list.

The first and second connection formats both rely on a list of connections following the instance declaration in an unit or platform instantiation. The two formats may in fact be mixed in the same connection, and connections may be left out of the list, either by the simple expedient of ending the list before the unit being instantiated runs out of ports

as on line 2, or by leaving entries in the connection list empty such as `instancename(,,)`.

The third connection format is shown on line 6 of Program 22, and unlike the first two is tied to the channel rather than the instance declaration. In this case a two part connection specifier of the form `{ X -> Y }` follows the channel declaration, where both X and Y must be dynamic identifiers which name ports. X and Y can either be ports on the unit in which the channel is instantiated in which case the dynamic identifier is simply the name of the port. Alternatively they may be name ports on child units, as on line 6 of Program 22 where the output of `OutChannel` is connected to the port `Value` on the instance `DisplayNumX`.

Dynamic identifiers in the third form of connection may include more than two parts. For example if the unit `IO::DisplayNum` a child unit, one could make a connection to a port on it from within `CounterExample` using a dynamic identifier like `DisplayNumX.Foo.InputPort`. This flexibility allows a designer to specify complicated connections without the overhead of changing every unit in the hierarchy. This ensures that what should be simple modifications to complex, many layered target systems, such as one might make to perform a simple RAMP experiment, are in fact simply to make.

These are referred to as explicit connections, and they are by far the most powerful, though the least familiar for HDL coders.

The arrow (`->`) in explicit channel connections may actual face either direction (`<-`) though the arrowhead always points to the output of the channel. When used for links, explicit connections are specified as a comma separated, arbitrarily long list of terminals, rather than a two element, arrow separated list of ports.

6.3.3 Arrays

Arrays of channel and link instances, like arrays of unit and platform instances use `[]` to specify the array bounds as part of the overall goal of creating parameterized simulations. While the instantiation of channel and link arrays is quite simple, making connections to them, as shown in Program 23, is more interesting. Program 23 is an expansion of Program 22 to specify an array of counters, each with their own input an output units.

Line 6 of Program 23 instantiates two arrays of channels, and specifies an explicit connection to the `OutChannel` array. The explicit connection list must specify an array of ports to connect to, and which ports to connect to which channels in the arrays. The `index#0` free variable essentially ranges over the bounds of the channel array, and is used as an index

Program 23 Multi-Counter

```
1 unit <width, size> {
2   instance IO::BooleanInput
        BooleanInputX[$size]([index#0]
        Value(InChannel[index#0]));
3   instance Counter<$width> CounterX[
        $size](InChannel[index#0],
        OutChannel[index#0]);
4   instance IO::DisplayNum<$width>
        DisplayNumX[$size];
5
6   channel InChannel[], OutChannel[$size
        ] { -> DisplayNumX[index#0].Value
         };
7 } CounterExample;
```

in to the `DisplayNumX` unit instance array. Thus line 6 specifies that `OutChannel[0] { -> DisplayNumX[0].Value }` all the way through `OutChannel[$size -1] { -> DisplayNumX[$size - 1].Value }` (though `$size - 1` is not valid RDL and is used for this explanation only).

Line 3 is relatively simple, and similar to line 6, though with positional instead of explicit connections. This means that `index#0` here ranges over the size of the `CounterX` unit instance array, which again happens to be `$size`.

Line 2 is more complicated, because of the interaction between named and array connections. In particular the snippet `[index#0]Value(InChannel[index#0])` specifies a connection between the `Value` ports on each element of the `BooleanInputX` array and the corresponding element of the `InChannel` array. The array bound appears before the port name, because the bound is on the instance array. The array bound would appear after the port name, if the connections were between an array of ports on one unit instance, rather than an array of units each with one port, and an array of channels.

Of course a mismatch in any of the array sizes for connections as in Program 23 this will result in a compiler error. This happens because an array of ports must be connected to an array of channels of the same bound. As a consequence, if the bounds of the port array are known, but the bounds of the channel array are not, the RDL2 parameter inference algorithm (see Section 5.1.7) will force them to be equal. Thus Program 23, on line 6 instantiates `InChannel` with the statement `channel InChannel[];` which specifies the array without specifying the bound. The connections on line 2 and 3 between the channel and unit arrays will both set the bound on this array.

For a more complicated example which motivated the index notation and array connections see

Section 7.1.

For all their value, array connections have some severe shortcomings. Particular they are limited to for-each constructs, and without parameter or index operators (such as modulo arithmetic) they are unable to describe simple structures like rings. In part this decision was based on the ability to write arbitrary compiler plugins (see Section 6.5), obviating the need for complex connections. However a large part of this decision was based on the realities of the compiler implementation (see Section 9), is being re-evaluated for the next revision (reference RDLC3).

6.4 Maps

RDL includes constructs to describe both hierarchical platforms and the mapping from a hierarchical target design to these platforms, in order to support target designs too large to be implemented on a single platform. Aside from the need to clearly specify interfaces, this is one of the primary reasons for the existence of RDL. While there exist some software tools for distributed systems, other hardware system description tools, such as EDK, shy away from multi-FPGA systems. RDL, by separating the target from the host, and capturing the details of both can support the automatic implementation of cross-platform designs.

In this section we discuss the RDL syntax which allows a designer to specify the mapping from units to platforms, in other words where units will be implemented. This is necessary because while RDL includes specifications for hosts and targets, it does not presume the existence of a reliable tool to automatically partition the target, though we have worked on such a tool (see Section 12).

6.4.1 Single Platform

A map in RDL is a specification of an RDF simulator or emulator implementation. A map specifies that a certain target system, denoted by a top level unit should be mapped to, and therefore implemented on, a particular host system, denoted by a top level platform.

As an example, Program 24 shows a mapping of the RDL "CounterExample" to two platforms: the Xilinx XUP FPGA board and to a Java Virtual Machine. A map declaration, such as `XUPMap` in Program 24, specifies a complete implementation which can be automatically generated by RDLC (see Section 8.4).

Program 24 CounterExample Maps

```
1 map {
2   unit CounterExample Unit;
3   platform XUP Platform;
4 } XUPMap;
5
6 map {
7   unit CounterExample Unit;
8   platform JVM Platform;
9 } JVMMap;
```

6.4.2 Cross-Platform

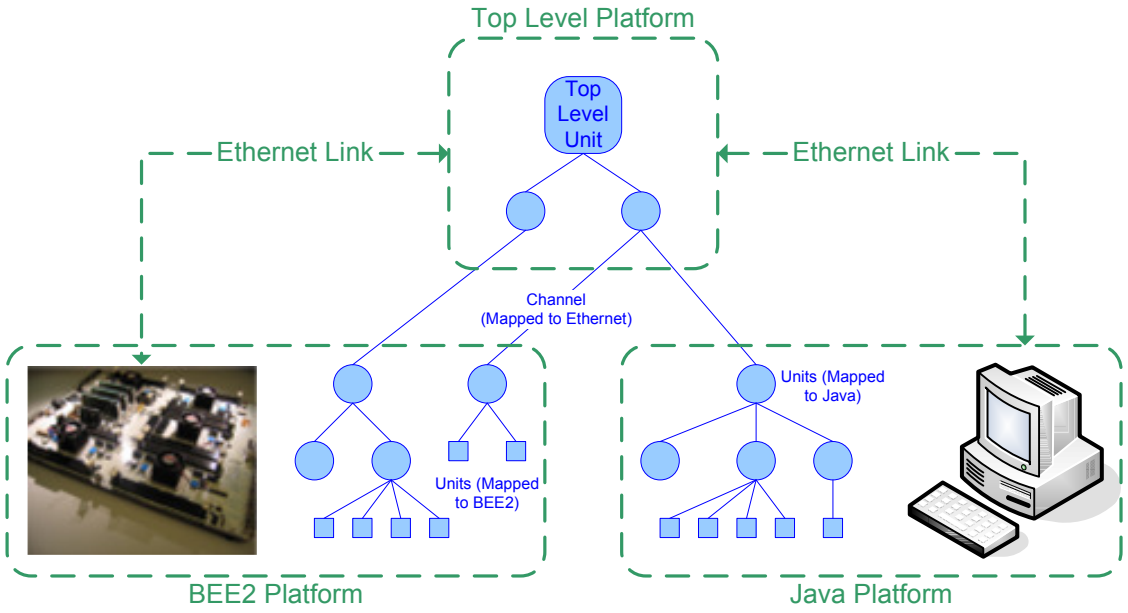
Basic mappings from a top level unit to a single platform have their place, but far more interesting are mappings to a complex hierarchy of platforms. Conceptually, a hierarchical mapping is quite simple, as shown in Figure 19 where a workstation platform is composed of a BEE2 and a desktop computer connected by Ethernet, and a complex target system is mapped to it.

A mapping from a unit to a hierarchical platform must also include more detailed mappings to specify the subplatforms on which subunits will be implemented. Furthermore, a mapping to a link must be specified for each channel connecting two units on different platforms, except in the case where there is only a single link between platforms. It should be noted that although RDLC2 requires these to be specified, we have worked out the algorithm and theoretical basis of a tool which will automatically partition a target system and map the relevant channels to links (see Section 12) though we preserve these languages features both because the tool is incomplete, and for more advanced RDL designers.

Shown in Program 25 is an example of a more complex mapping of the "CounterExample" to a pair of CaLinx2 [33] boards, with the input and counter on the first board and the display on the second board. Lines 2 and 3 specify that we are mapping from the `CounterExample` unit to the `DualCaLinx2` platform. Lines 5-9 specify a mapping of instances of the `IO::BooleanInput` and `Counter` units to an instance of the `CaLinx2` platform. Lines 10-13 specify a mapping of an instance of the `IO::DisplayNum` unit to an instance of the `CaLinx2` platform.

Together the `Map0` and `Map1` mappings contain all the units instantiated in `CounterExample` and `DualCaLinx2`. However, what remains is to specify which platform instances from `Platform` correspond to which platforms in the mapssubmaps. For this example it may be quite obvious to the human reader, but in more complex examples, for example

Figure 19 Mapping to Platforms: BEE2 and Java



with many instances of the same unit and platform the map statements on lines 15-23 are vital.

For another example of a cross-platform map, please see Section 7.4.

As with a simple mapping to a leaf platform, a mapping to a hierarchy of platforms will allow RDLC to produce all of the necessary output to instantiate and connect the various leaf units, which have been implemented in a host language. The details of this process and the resulting code are discussed elsewhere (see Section 8.4). It is interesting to note that the ability of RDL to express these mappings allows a separation from the heuristic optimization problem of generating a mapping from the more concrete task of implementing one.

In addition to the requirements imposed by real platforms (*e.g.* that links may not be point to point like channels) platform specification is complicated by the desire to reduce compilation cycles on very large systems. For example a large design filling a Xilinx V2Pro 70 FPGA on the BEE2 [25, 37] may take many hours to place and route, forcing RDLC to support some form of compile-once, run-many to reduce overall target implementation costs. Because maps, like units and platforms can be specified hierarchically, it is possible for a design to consist of many instances of identical mappings. Because each mapping specifies the complete set of units and the platforms, this means that two identical mapssubmaps will be identical even when run through whatever hardware or software compilation tools are necesary. This in turn means that these tools, which are slow, need only be run once.

Program 25 DualCaLinx2 CounterExample

```
1 map {
2   unit CounterExample<32> Unit;
3   platform DualCaLinx2    Platform;
4
5   instance {
6     unit IO::BooleanInput BooleanInputX
          ;
7     unit Counter          CounterX;
8     platform CaLinx2      CaLinx2;
9   }                       Map0;
10
11  instance {
12    unit IO::DisplayNum   DisplayNumX;
13    platform CaLinx2      CaLinx2;
14  }                       Map1;
15
16  map Unit.BooleanInputX onto Map0.
      BooleanInputX;
17  map Unit.CounterX onto Map0.CounterX;
18  map Unit.DisplayNumX onto Map1.
      DisplayNumX;
19
20  map Unit.CounterX.Count onto
      Platform.Board[0].UARTTerminal;
21
22  map Platform.Board[0] onto Map0.
      CaLinx2;
23  map Platform.Board[1] onto Map1.
      CaLinx2;
24 } DualCaLinx2CounterExample;
25
```

While this requires a good deal of specification on the part of the RDL writer, schemes which place a higher burden on RDLC are infeasible in the short term. As it is, the compiler cannot automatically partition a design, or cope with multi-hop channel to link mappings. Even without these features, RDL provides an excellent framework for research in these areas by providing a common specification language, allowing a partitioning or network embedding tool to handle the complex algorithms, and leave the implementation to RDLC. It should be noted that the need to minimize compile times by generating a regular mapping is one of the reasons the RDL mapping problem differs from conventional ASIC or FPGA placement and routing. The theoretical work in this area is addressed elsewhere (see Section 12).

6.4.3 Summary

In order to support target designs too large to be implemented on a single platform, RDL includes constructs to describe both hierarchical platforms and the mapping from a hierarchical target design to these platforms. In this section we have we have discussed the RDL syntax which allows a designer to specify the mapping from units to platforms, in other words where units will be implemented. The ability to map large target designs on to complex, hierarchically constructed hosts is one of the primary reasons for the existence of RDL.

6.5 Plugins

RDL2 provides a powerful plugin mechanism, whereby developers prepared to interact trivially with RDLC can in turn get access to the Abstract Syntax Tree (AST) of the RDL description being compiled. This allows for expansion, and integration of external tools with RDL, features proved useful in all applications thus far (see Sections 13, 14 and 15).

With the tcl-ization of RAMP Blue [79, 63, 84, 64] and the level of parameterization required of prospective RAMP designs, it is clear that the RDL parameterization mechanism must be Turing-complete. Furthermore, RDLC must generate code in a variety of output languages, while providing forward compatibility with new platforms, and therefore new types of links. In order to provide this first, a Turing-complete scripting language could be provided within RDL, at a high cost in man hours and complexity. However, only an external language, more focused on software development would provide the code and link generation required, thus prompting the RDL plugin architecture (see Section 10).

In this section we will focus on the lexical, syntactic and semantic issues of RDL plugins. There are several kinds of plugin declarations, and though they share a base syntax as shown in Program 26, they are treated differently at the language level. Plugins may be specific to the back end of the compiler flow, that is to a particular language or platform as shown in Section 6.5.2. Plugins may also denote terminal types (see Section 5.3.2) or they may be entirely general as shown in Section 6.5.1. Section 10 provides a complete description of the plugins we have implemented to date.

6.5.1 Front End

We use the term "general" or "front end" to describe those plugins which are designed to affect the beginning of the RDL compilation process. These include plugins which *e.g.* generate a unit netlist (see Section 13.2.4) or modify some parameters (see Section 10.3.2). In other words these plugins have arbitrary access to RDLC's AST and therefore may modify the current description, as needed.

Program 26 Plugin Invocation

```
1 unit {
2   plugin "Dummy" DummyInvocation;
3 } AUnit;
```

As shown in Program 26, any statement beginning with the keyword `plugin` is a general plugin invocation. These invocations take the form `plugin "string" invocationname;`, where the string, double quotes, gives the name of the plugin to be invoked. During compilation a plugin invocation statement of this form will cause RDLC to load and run the specified plugin (see Section 9.3), thereby allowing it to execute arbitrary Turing-complete code with full access to the compiler data structures, particularly the AST. The details of this process are explained elsewhere, suffice to say that the interactions between plugins, parameter inference and the guarantee that declaration ordering does not matter while complex to implement, are necessary to ensure RDL is easy to understand and write.

Plugins, like most things which can be instantiated in RDL, can have parameters. What makes plugins unique is the fact that they are responsible for their own parameter validation, meaning that they're formal parameter list is not part of the RDL source code. As shown in Program 27 plugin parameter actual values use the standard syntax, and

are applied to the string as it represents the "type" of the plugin.

Program 27 Plugin Parameters

```
unit {
  plugin "Dummy"<"ignored", 0, ::0::
      AUnit> DummyInvocation;
} AUnit;
```

Plugins invocations will sometimes need to cooperate, for example a plugin responsible for generating a network switch will need to cooperate with a plugin responsible for the overall network topology. The RDL parameter inference mechanism allows parameter values to be propagated both up and down an instantiation hierarchy, effectively making this quite simple. A parameter can be declared on a unit, passed to two plugins, and never given a value in the RDL source. One plugin may then assign a value to this parameter, which the other plugin will then be able to use. This complex detail of the parameter inference algorithm in RDL has been put to good use in the basic set of plugins already.

For a complex example involving a number of plugin invocations, please see Section 7.3.

6.5.2 Back End

In contrast to "front end" plugins which allow for RDL generation, "back end" plugins allow for arbitrary code to specify particular implementation details. In particular, while the most valuable units from the research perspective will be portable across platforms by design, library units representing I/O blocks at the target level are inherently not portable. Rather than declare completely separate units for these common I/O blocks, RDL plugins can be parameterized by the platform or language, the compiler back end to which they apply.

Program 28 shows examples both of language specific and platform specific plugin invocations of the form `plugin qualifier "string" invocationname;`. In this syntax the *qualifier* must be either an RDL recognized language, or the static identifier of a platform. In either case, the plugin will not be invoked until the code generation stage of compilation, meaning that while it has no chance to modify the RDL AST, it will have access to the code generation facilities.

In essence the *qualifier* forms a platform and language-based switch statement, allowing the unit designer to specify a number of similar, possibly overlapping unit implementations. This facility has worked effectively for a number of applications, and

Program 28 Back-End Plugins

```
unit {
  plugin Verilog "Dummy" Language0;
  plugin VHDL "Dummy" Language1;
  plugin Java "Dummy" Language2;

  plugin Platforms::ModelSim "Dummy"
      Platform0;
  plugin Platforms::XUP "Dummy"
      Platform1;
  plugin Platforms::SunJVM "Dummy"
      Platform2;
  plugin Platforms::DE2 "Dummy"
      Platform3;
} FIFO;
```

yet the awkwardness of it has prompted us to evaluate alternatives for future work (see Section 16.4.1).

The *qualifier* in a plugin invocation is what denotes the invocation as being "back end", and what limits when the plugin is actually invoked. Invocations without this qualifier are assumed to be "front end" invocations, and though there is some overlap it should be considered a relatively special case as it is difficult to program such plugins (see Section 10.3.2).

6.5.3 Summary

RDL plugins are a powerful plugin mechanism which allows expansion, and integration of external tools with RDL. In this section we have presented the lexical, syntactic and semantic issues of RDL plugins. Plugins provide per-platform and per-language unit customization for I/O library units, as well as RDL AST access for complex generators written in a Turing-complete software language, rather than a complex, rdl-specific scripting language. Finally, as described elsewhere (see Section 5.3.2), plugins are the basis of automatic link generation (see Section 10.4.1), one of the primary goals of RDLC and therefore RDL.

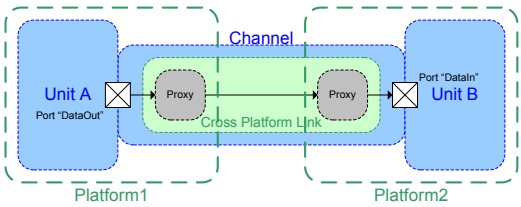
6.6 Advanced RDL

This section covers some novel uses of RDL which are possible thanks to its unique design in contrast to the main RDL language documentation which covers the basic capabilities of RDL their syntax. The examples in this section are provided primarily to open the readers mind to the range of uses for RDL beyond simple RAMP simulations.

6.6.1 Zero Latency Channels

As is discussed elsewhere (see Section 4.3.3) it is an easy matter to implement links within an FPGA or software which have a 0 host cycle latency. Of course these links in turn allow us to implement 0 target cycle latency channels, an intriguing prospect for some potential RDL users, even if it is disallowed by RDF (see Section 3.6). What is frustrating is that it is difficult to imagine how these channels might be implemented when a target design is mapped to a more complex platform, composed of *e.g.* multiple FPGAs connected by a packet switched network.

Figure 20 Zero Delay Channel



RDL provides a clean, easy to use abstraction of cross-platform communication in such a way that the communication can be abstracted from its implementation. While implementing a zero latency channel we would like to retain this generality.

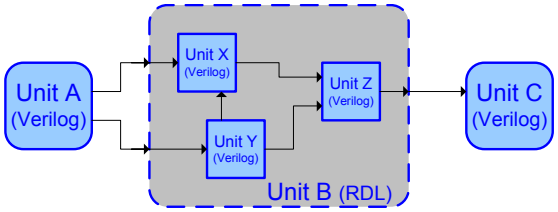
Figure 20 shows one possible, and quite elegant solution which leverages the generality of RDL in a new way. The link implementation is itself specified in RDL. The compiling the complete design would include two separate invocations of RDLC, one for the main target and host, and one simply to generate the cross-platform communications. Thus the lower level RDLC invocation abstracts the physical communication which the higher level RDLC invocation can add additional logic to, to create the proper timing.

The alternative, that every link generator plugin (see Section 10.4.1) be able to deal with the complex simulation control logic to create 0 latency channels, would unnecessarily complicate the lives of link generator implementors.

6.6.2 RDL in RDL

Because RDL is a system level language with support for emulation it is possible to use it for implementation rather than simulation. Building on the previous example, of a zero latency channel, one could imagine building entire subsystems of a design using RDL as shown in Figure 21. For example a host level network (see Figure 17) has been suggested as an interesting way to increase the generality of some platforms with incomplete connectivity.

Figure 21 RDL in RDL



RDL in RDL could also be used for simulating the simulation, allowing the researchers designing key components of the RDL infrastructure to simulate their own work before implementation.

Also possible would be using an RDL emulation as a component of an RDL simulation. For example one might use existing RDL units to construct a functional emulation of a subsystem to which one would like to apply a simple, monolithic timing model. This subsystem could be run through RDLC to generate implementation code, to which some timing model code is added to create a more abstract unit implementation.

The power of RDL in RDL stems from the fact that RDL is both a simulation and emulation language. While the main focus of RAMP is on simulations, the ability to ignore the timing logic (see Section 4.2.4) and run at full speed is easy to implement at the language and compiler level, and allows the RDL parameterization and link generation to be used for simple system construction. Of course there are other system level languages in existence, RDL is by no means unique in this respect, but for a project already using RDL for simulation, it is attractive to reduce the number of languages involved.

6.7 Conclusion

The main goal of RDL is to clearly describe both parameterized target (see Section 3) and host (see Section 4) systems, allowing one to succinctly map between them. In this section we laid out the syntax and semantics for the declaration of units & platforms and channels & links, including the assembly of these to describe complete systems.

RDL provides an abstraction of the locality and timing of communications, enabling timing accurate simulation and cross-platform designs. RDL is a combination of a structural modeling language and a simple netlisting language for dynamic, or "run time" systems, such as a subset of Verilog or VHDL might provide. RDL does not, and will

never, provide for the specification of leaf unit behavior, as it is aimed at tying together existing designs, and there are enough behavioral languages in common use. Unit designers must produce the gateware (Verilog for RDLC1 & 2) or software code (Java for RDLC1) for each unit in their chosen language, and specify, in RDL, the types of messages that each input or output port can carry as well as the structure of the target and host systems.

The biggest cost associated with using RDL is the time taken to describe a design using target model and RDL. Second, is the area, time and power to implement this model, all of which are easily controlled and are functions of the simulation rather than the language. This is a major benefit of RDL above other system level languages, as it does nothing to obstruct the skilled hardware designer.

The biggest benefits of RDL are: deterministic timing, clean interface encapsulation, implementation language independence and the sharing of units between researchers that these things allow.

In this section we have presented those aspects of RDL which are crucial to the specification of both target (see Section 3) and host (see Section 4) systems. We have drawn a number of examples of RDL syntax from the "CounterExample" (see Section 7.4), which is a complete RDL description of a very simple system built around a counter. We have used these examples to motivate a discussion not only of the exhibited language syntax but also the reasons for its creation. In concert with a description of the RDL type system (see Section 5.2), this provides a clear picture of the primary features of RDL: unit and platform netlisting.

Chapter 7

RDL Examples

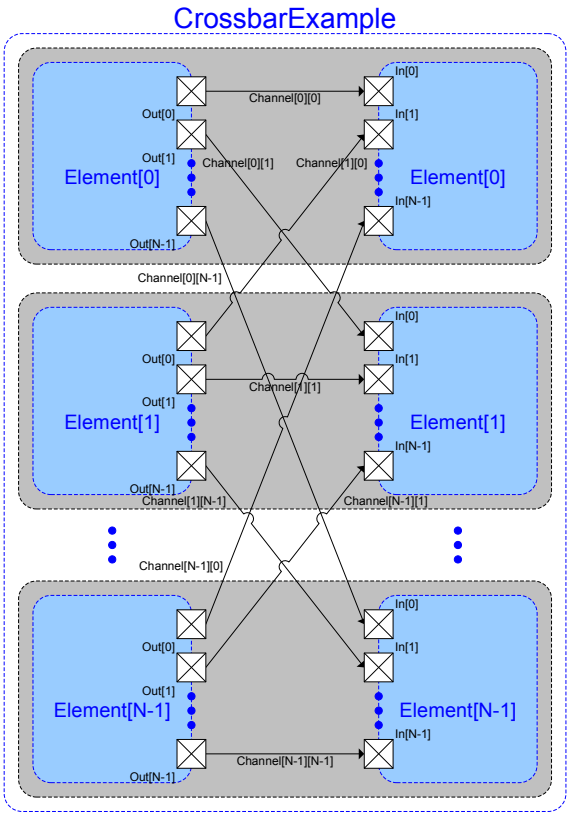
One of the easiest ways to learn any new computer language, particularly those which are dissimilar to those within the reader's experience, is by example. In this section we describe, in detail several examples of RDL ranging from what are essentially compiler test cases (sections 7.1 and 7.2) to complete RDL systems (sections 7.4 and 7.6). Most of these examples are available in a more complete, source code form along with RDLC itself from [9]. The examples in this section are small and easily understandable, and are not commensurate with genuine applications (see Sections 13, 14 and 15) of RDL which are expected to be at least an order of magnitude more complex.

7.1 Crossbar

Program 29 and Figure 22 show the most powerful use of parameters and inference combined to build a crossbar of channels. The goal of this example, originally created as an RDLC2 test case (see Section 9.6), is to declare an array of `Elements`, and connect them with an all-to-all crossbar or channels. This is accomplished in 2-4 lines of RDL, using instance and channel arrays in conjunction with indexed connections (see Section 6). Furthermore, by using parameter inference, we have reduced the specification of the bitwidth and number of elements to a single point, line 7, allowing the remaining values to be inferred.

Walking through the code, lines 1-4 of Program 29 declare simple unit `Element` which has two port structures, both of them arrays, one for input (`In`) and one for output (`Out`). `Element` also has two parameters, one for the number of ports (`NPorts`) and one for the bitwidth of these ports (`Width`). Note however, that the parameters are each used once, meaning that the bitwidth of the messages sent through the `In` ports is not specified in this code and must be inferred. Similarly, the bound on the `Out` port array is unspecified and must be inferred.

Figure 22 Crossbar



Lines 7-9 of Program 29 are the most interesting. Line 7 instantiates a two dimensional array of `Elements`, and uses indexed, positional connections to connect them to the `Channels` array instantiated on line 10. Line 9 in essence is a straight-through connection from each `Element` and each `Out` port to a channel with the same indices.

The crossbar is created by virtue of the change between `index#0` major array selection and `index #1` major array selection on `In` port array on line 8. Thus while line 9, creates the connections $\forall x, y\ Out[x][y] \rightarrow Channel[x][y]$ line 8 creates the

Program 29 Crossbar.rdl

```
1 unit <NPorts, Width> {
2   input bit<>[$NPorts] In;
3   output bit<$Width>[] Out;
4 } Element;
5
6 unit {
7   instance Element<2, 8>
        Elements[]
8     ([index#0][index#1](Channels[index
          #1][index#0]),
9      [index#0][index#1](Channels[index
          #0][index#1]));
10  channel Channels[][];
11 } CrossbarExample;
```

connections $\forall x, y\ Channel[y][x] \rightarrow In[x][y]$ effectively creating the crossbar shown in Figure 22.

7.2 CrossPlatform

The goal of the RDL description shown in Figure 23 and Program 31 is to test all of the corner cases of channels and links crossing unit and platform boundaries respectively. This makes it an ideal example of exactly how complex mapping can be become, though the instance names are meaningless letters.

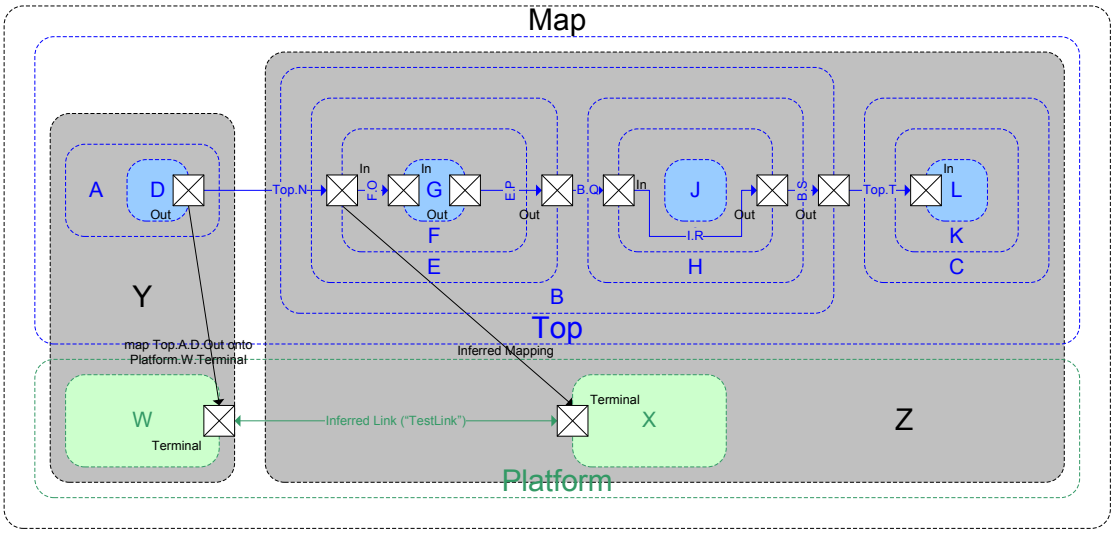
The design consists two platforms `W` and `X` which have been connected to form platform `Platform` on to which the unit `Top` is mapped by the map `Map`. What makes the design interesting is that the unit instances within `Top` have been split between `W` and `X`. To accomplish this, unit `A` is mapped to `W` by the map `Y` and `B` and `C` are mapped to `X` by map `Z`. These two mappings are assembled by `Map` and given a correspondence to the higher level `Top` on lines 36-38 and `Platform` on lines 40-41.

In addition to the mapping of units to platforms, this example shows how to map channels to links. In particular channel `N` instantiated on line 6, is mapped to the link between `V` instantiated on line 20, by the statement `map Top.A.D.Out onto Platform.W.Terminal;` on line 39. This particular statement specifies a mapping from the port `Out` on the unit instance `Top.A.D` This mapping actually specifies that the `Out` port is mapped to the `Terminal` terminal, with the desired channel to link mapping implied by this. Technially the statement on line 39 is unnecessary as there is only one link between `W` and `X`, but it is included here to complete the example.

Program 30 CrossPlatform Units

```
1 unit {
2   instance {
3     output bit<32> Out;
4   } D;
5 } A;
6
7 unit {
8   instance {
9     instance {
10      channel O { In -> G.In };
11      input bit<32> In;
12      instance {
13        input bit<32> In;
14        output bit<32> Out;
15      } G;
16    } F;
17    channel P { F.G.Out -> Out };
18    output bit<32> Out;
19  } E;
20
21  channel Q { E.Out -> H.I.In };
22
23  instance {
24    instance {
25      input bit<32> In;
26      instance { } J;
27      channel R { In -> Out };
28      output bit<32> Out;
29    } I;
30  } H;
31  channel S { H.I.Out -> Out };
32  output bit<32> Out;
33 } B;
34
35 unit {
36   instance {
37     instance {
38       input bit<32> In;
39     } L;
40   } K;
41 } C;
```

Figure 23 Cross Platform Mapping



7.3 FIFO

Though RDL makes the declaration of a FIFO unit rather unnecessary by subsuming such functionality in the channel model (see Section 3.3) we present this unit as an easily comprehensible example. Furthermore, this unit proved quite useful before complete link generators were available to support the channel model, and was used in several of our sample applications (see Sections 13 and 14). This example is not only meant to be understandable, but also to show off how back end builder plugins, and some of the grittier details of RDL can be combined to good use.

Shown in Program 32 is the RDL declaration for the FIFO unit.

Most notable, the unit has two ports `Input` and `Output` whose types are specified by the parameter `Type`, and which are marked `opaque`. Together this means that the FIFO implementation neither knows nor cares what kinds of messages pass through it. In fact the `Type` parameter need never be specified during instantiation, as parameter inference will ensure both ports have the same type, and that it matches the instantiation context. The opaque keyword on the other hand disables message packing and unmarshaling (see Section 4.2) so that the FIFO implementation may be of minimum width, and needn't in any way know the structure of the message types (see Section 5.2).

Setting aside the complex RDL specification for the ports, lines 5-7 of Program 32 specify the implementation of this unit using three RDL plugins. The plugin `"FIFO"` is invoked on line 6 and the plguin `"FIFOUnit" on line 7, whenever the FIFO is \glslink{map}{mapped} to a \gls{platform} which uses the \lstinline[language=RDL]`Verilog! language. These plugins create the FIFO control logic, and interface it to the inside edge respectively.

Line 5 is more interesting in that the exact plugin to invoke is specified not by a constant, but by the parameter `MemType`. This allows the exact nature of the memory generated to be dependent, in this case, on the platform, a step necessary to ensure efficiency of larger FIFOs. Lines 9-13 are responsible for setting the `MemType` parameter to the name of the correct memory generator plugin.

All of the plugins used in this example are described more completely in Section 10.

7.4 Counter Example

Shown in Program 33 is the primary example for new RDL developers, the humorously named "CounterExample[1]." This example, and snippets of the complete code (not reproduced here) are used to illustrate the basic features of RDL and the compiler. This section is geared towards a hardware demonstration of RDL2, and expects the reader is familiar with the basics of FPGA PAR and HDL simulation tools.

The counter example is a simple RDL description of a 32bit Up/Down counter which will count up by one each time it receives an input message that is an `1` and down by one each time it receives a `0`. This counter will produce an output message

[1] This name is a perfect example of computer science humor in that it has confused many readers. We apologize.

Program 32 FIFO.rdl

```
1 unit <Type, Depth, MemType, AWidth> {
2   opaque input $Type Input;
3   opaque output $Type Output;
4
5   plugin Verilog $MemType <$AWidth, "Input", ""> Memory;
6   plugin Verilog "FIFO" <$Depth, $AWidth> FIFO;
7   plugin Verilog "FIFOUnit" <"FIFO", "Memory", "Input", "Output"> FIFOUnit;
8
9   plugin ::1::Platforms::ModelSim "SetParam"<$MemType, "ModelSimMemory">
        SetMemModelSim;
10  plugin ::1::Platforms::XUP "SetParam"<$MemType, "Virtex2ProMemory"> SetMemXUP
        ;
11  plugin ::1::Platforms::S3 "SetParam"<$MemType, "Spartan3Memory"> SetMemS3;
12  plugin ::1::Platforms::CaLinx2 "SetParam"<$MemType, "VirtexEMemory">
        SetMemCaLinx2;
13  plugin "SetParam"<$MemType, "Dummy"> SetNoMemory;
14 } FIFO;
```

consisting of its count value *after* the appropriate action is taken upon receipt of each input message. Of course if it receives no input, it will produce no output. This can be summed up in a simple state transition diagram such as in Figure 25.

While this is a simple example, and smaller than a typical unit (particularly for an RDF design), it illustrates the basics of RDL. The `::Counter` is declared to accept unstructured 1-bit messages at its port "UpDown" (`::Counter.UpDown`) and produce 32-bit messages at its output port "Count" (`::Counter.Count`). Of course this is a leaf unit, which will be implemented directly in the host language (Verilog or Java for example).

RDL and the compiler also support hierarchically defined units like `CounterExample` in this code snippet. Inside this unit, there are two channels, shown without detailed timing models, which are used to connect the three unit instances. This example also shows all three styles of port connections named (line 3), positional (line 5) and explicit (line 11). Explicit connections can use qualified dynamic identifiers to specify connection of a local channel to a port significantly lower in the hierarchy, without explicit pass-through connections at each level making debugging and modification for test much easier.

This section includes a brief description of each of the units in the counter example. Shown in Figure 24 is a diagram of the overall structure of the counter example system. Please note that the counter example source code can be downloaded from the RAMP website [9] in the `examples/counterexample` directory inside of the RDLC2 distribution zipfile.

7.4.1 Unit: `CounterExample`

This is the top level unit of the design, as may be noted by its lack of input and output ports. This unit instantiates the other three and connects them together as shown in Figure 24.

In addition to the examples of unit instantiations shown in the code for this module, you are encouraged to examine both the channel declarations and, more interestingly, the port-channel connections. There are three ways to connect a port to a channel and all of them have been shown in this unit.

In addition to the two commonly used in Verilog (named and positional), the third and most interesting is on line 11: `Channel OutChannel { -> DisplayNumX.Value };`, which is a channel with a single connection. The source of this channel is connected elsewhere, but here the destination connection is specified as the input port `Value` on the unit instance named `DisplayNumX`. This method of connections, while slightly more verbose than the other two allows a connection to be made at a high level of the instance hierarchy, without declaring and connecting channels at each intermediate level. This is particularly useful for debugging.

In addition to the port, channel and unit instances, there are several plugin invocations in Program 33. A plugin invocation starts with the keyword plugin, followed by a string literal specifying the plugin to run, and finally an name for this plugin invocation. Note that plugins may also accept parameters, and be limited to run only for specific platforms or platforms which generate code in specific languages.

As an example here is one of the plugin invo-

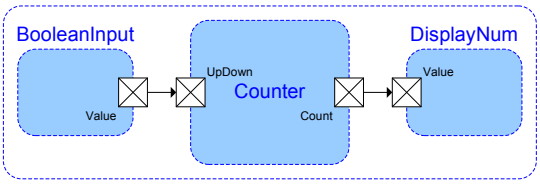
Program 31 CrossPlatform.rdl

```
unit {
  instance A A;
  instance B B;
  instance C C;

  channel N { A.D.Out -> B.E.F.In };
  channel T { B.Out -> C.K.L.In };
} Top;

platform {
  language "Verilog";
  plugin "TestLink" DefaultLink;
  terminal ("TestLink") Terminal;
} Verilog;

map {
  unit Top Top;

  platform {
    link V { W.Terminal, X.Terminal };
    instance Verilog W;
    instance Verilog X;
  } Platform;

  instance {
    unit A A;
    platform Verilog Verilog;
  } Y;

  instance {
    unit B B;
    unit C C;
    platform Verilog Verilog;
  } Z;

  map Top.A onto Y.A;
  map Top.B onto Z.B;
  map Top.C onto Z.C;
  map Top.A.D.Out onto Platform.W.
      Terminal;
  map Platform.W onto Y.Verilog;
  map Platform.X onto Z.Verilog;
} Map;
```

Program 33 CounterExample.rdl

```
unit <width> {
  instance IO::BooleanInput
    BooleanInputX(Value(InChannel));
  instance Counter<$width>
    CounterX(InChannel, OutChannel);
  instance IO::DisplayNum<$width>
    DisplayNumX;

  channel InChannel;
  channel OutChannel
    { -> DisplayNumX.Value };
} CounterExample;

unit <width = 32, saturate = 1> {
  input  bit<1>       UpDown;
  output bit<$width> Count;
} Counter;

unit {
  output bit<1> Value;
} BooleanInput;
unit <width = 32> {
  input bit<$width> Value;
} DisplayNum;
```

Figure 24 Counter Example Block Diagram



cations: `Plugin Platforms::ModelSim "SetParam"<$width, 32> ModelSimWidth;`. This declares an invocation of the `SetParam` plugin, which we name `ModelSimWidth`. To this plugin, we are passing two parameters: `$width` and `32`. Finally, the plugin is restricted to run only when `CounterExample` is mapped to the `Platforms::ModelSim` platform.

The purpose of this plugin invocation is to set the parameter `$width` to `32`. In general the `SetParam` plugin (see Section 10.3.2) will set its two arguments equal, and fail silently if the first argument already has a value. In conjunction with the platform or language restrictions allowed on plugin invocations, this allows `SetParam` to form an effective "select" or "case" statement for setting platform dependent parameters, where the silent failure allows it to be used for the "default" case as well. Note that RDLC2 also includes a `ResetParam` plugin which will forcibly set the first argument, rather than silently failing.

7.4.2 Unit: `Counter`

The core of this example is this simple `Counter` unit. Designed as an up/down counter which is enabled by the receipt of a message at its `UpDown` port, this counter in response to the input counts and sends the new value out on it's `Count` port. Shown in Table 1 is a list of the relevant port information in a more immediately recognizable form than RDL, and Figure 25 shows the state transition diagram for the counter.

Table 1 Counter Ports

Dir	Width	Name	Description
In	1	UpDown	Counter up/down enable messages
Out	width	Count	New count value message output
Param	-	width	The bitwidth to use for the count
Param	-	Saturate	Should this counter saturate?

We highly recommend that you refer to both the RDL declaration for this unit, in `CounterExample.rdl` and its Verilog implementation in `Counter.v`.

As a final note, the `width` parameter listed in Table 1, is set in `CounterExample` through the use of the `SetParam` plugin. Because plugins can be platform or language specific, this allows the width of counter to vary from platform to platform. For example on the Xilinx XUP board, the counter will be 4 bits wide to match the four LEDs, whereas in ModelSim, the counter will be 32 bits wide.

7.4.3 Unit: `IO::BooleanInput`

This is a simple unit which produces messages in response to the push of a button. The value of these messages is decided by a switch on the board. It should be noted that the fact that this unit is declared in the `IO` namespace is not important to the language or compiler as namespace names are only for human consumption.

Of key interest in the RDL declaration of this unit are the platform specific plugins and parameters, used to declare the existence of board-level signals which are not part of the RDL target model. In this case there are two inputs declared: `_BTN` and `_SW`, both of which are given a bitwidth of `1`, and external pin location constraints to connect them to the proper signal on the board. Of course, this is the reason for four separate declarations for each, one for the XUP, one for CaLinx2, one for the DigilentS3 and one for the Altera DE2 board.

Table 2 BooleanInput Ports

Dir	Width	Name	Description
Out	1	Value	The value of the switch, sent when the button is pushed

Again, we refer the reader to the RDL source code and `BooleanInput.v`. Notice that the Verilog implementation of this unit takes advantage of a series of pre-existing Verilog modules for debouncing and edge detection to clean up the signal from otherwise noisy real world switches.

7.4.4 Unit: `IO::DisplayNum`

This unit has been designed to output a variable width message on the CaLinx2, XUP, DigilentS3 and Altera DE2 boards. Of course some boards have more or fewer LEDs, meaning that the most significant bits may be truncated. For example there are only 4 LEDs readily usable on the XUP board, so only the lowest four bits of the message are displayed.

Table 3 DisplayNum Ports

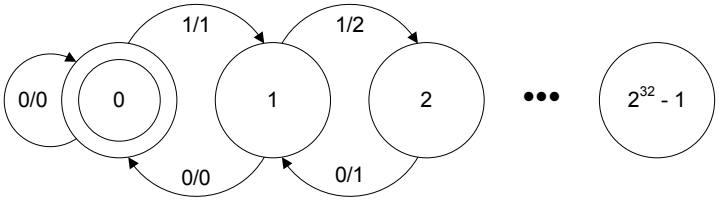
Dir	Width	Name	Description
In	width	Value	The value to be displayed
Param	-	width	The bitwidth of the input

The combination of the platform specific plugins, and some slight of hand in the Verilog implementation in `DisplayNum.v`, allowed us to get away with a single implementation for a variety boards. In general, we believe that platform specific libraries of I/O units may well be necessary. Bear in mind however that only real world I/O is visible to the user of RDL in this way, all unit to unit communications, even in the event that they leave the FPGA, board or computer, are handled through RDL and RDLC.

7.4.5 Platforms

A platform declaration has two key parts: a language and series of plugins which specialize the language to generate runnable host level systems. In

Figure 25 Counter State Transition Diagram



general platforms may be hierarchical, for example a researcher with an XUP board connected to a BEE2 and his workstation, would create a higher level platform which instantiated those three and described the physical connections (links in RDL paralance) between them.

The language declaration in a platform will determine which language designs for this platform should be generated in. The counter example is designed to be instantiated in Verilog at the current time.

Most of the useful platforms in this lab have additional plugins. For example the Verilog language back end will look for plugin instances called `Engine` and `Library`, as these have special meaning. In addition, most of the platforms in `CounterExample.rdl` include a plugin instance which will run a back-end toolflow such as ModelSim, XFlow or Quartus.

The engine plugin is primarily for generating the firmware which will generate clock and reset signals, and is therefore often board specific. For example the ModelSim engine simply uses an initial block to fake these signals, whereas the Xilinx and Altera engines use clock buffers and shift registers to generate automatic resets.

Similarly the library plugin is used by many pieces of the back end, including the engine, to load platform specific modules and other pieces of code, Verilog in this case, which has been hand-written into a library. In the future a number of standard pieces of gateware, such as multipliers and counters will be included in this library.

Finally all language back ends will honor, if it is found, a plugin invocation named `DefaultLink` which indicates the link generator plugin to be used for those channels what have not been mapped to a specific link, and for those links whose type is unspecified. In the XUP platform, there are two declarations and a short note about this, which more advanced users will find helpful.

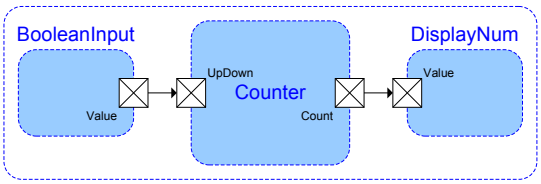
7.4.6 Mappings

A mapping specifies which platform a given unit, or tree of units should be implemented on. Because RDLC does not manage automatic design partitioning, hierarchical maps will be required for hierarchical platforms. There are several mappings for the counter example, most of which are to single FPGA platforms or HDL simulators.

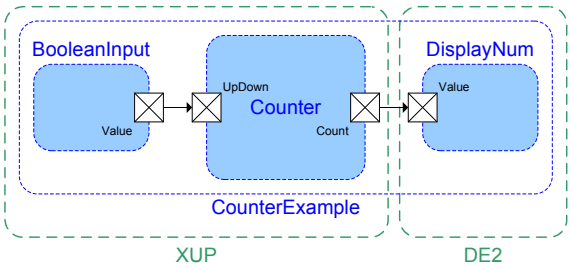
There are also two more interesting mappings in `CounterExample.rdl`: `::Maps::DualCaLinx2` and `::Maps::XUPDE2`. Both of these will map the CounterExample two a composite platform, the first consisting of two CaLinx2+ boards with their COM1 ports tied together, and the second consisting of an Xilinx XUP and an Altera DE2 board connected through their serial ports. Note that for the serial port connections you will need a "null modem" cable with male connectors at both ends (a null modem cable with two gender changers, or a standard cable with a null modem adapter should do).

Figure 26 Cross Platform Counter Example

(a) Target



(b) Implementation



Both of these mappings will put the `BooleanInput` and `Counter` units on one board, and the `DisplayNum` unit on the other.

7.4.7 Counter Example

In this section we have given the code, and a detailed explanation of the counter example, the primary RDL example for those new to the language and tools. The complete code for the example, as well as instructions on how to get it working are included with the RDLC2 source code which is downloadable from [9].

7.5 CPU Example

The counter example in the previous section is clearly overly simple for a RAMP unit, as useful as it is as an example. Because RAMP units must be latency-insensitive, in general they will be relatively large components of a design, to use the original examples (see Section 2.4), a processor with L1 cache, a DRAM controller or a network controller. In this section we present yet another RDL example, Program 34, which illustrates the declaration of three components: a processor, a cache (presumably L2 or lower) and a memory controller of some kind.

The memory messages are declared on lines 2-9 to deal in bursts of 256 bits or 32 bytes, and is byte addressable with a 32 bit address. As such a burst of data is 256 bits, and the corresponding address is $32 - log_2(256/8) = 27$ bits. A `Store` then is a structured message containing some `BurstData` and the `BurstAddress` at which to write it back. Loads are split-phase in this example with two separate messages: a `LoadRequest` and a `LoadReply` which are clearly nothing more than new names for `BurstAddress` and `BurstData` respectively.

After all of the relevant structured messages are declared, there are declarations for the two union messages required for this example. The two ports are declared called `MemIn` and `MemOut`, where `MemOut` carries simple `LoadReply` messages. The `MemIn` port declaration is only marginally more complicated as it is a union capable of carrying both `Load` and `Store` messages, along with a tag indicating which the current message is. Tags in RDL can automatically assigned in a deterministic and repeatable manner, ensuring that they will not change so long as the union message declaration remains unchanged or they may be explicitly specified.

Lines 11-19 also declare a port structure, an interface, called `MemIO` consisting of memory input and output ports. This interface allows the `CPU::CPU`, `CPU::Cache` and `Memory::Memory` units to easily declare their support for the complete memory request and response interface.

Program 35 completes this example by declaring a top level unit `System` which instantiates the

Program 34 A CPU and Memory Model in RDL

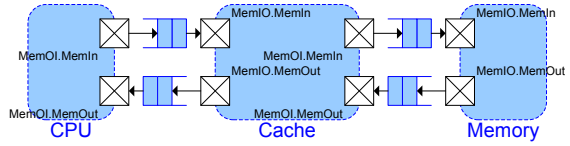
```
1  namespace {
2    message bit<256>      BurstData;
3    message bit<27>       BurstAddress;
4    message mstruct {
5      BurstAddress        Address;
6      BurstData           Data;
7    } Store;
8    message BurstAddress LoadRequest;
9    message BurstData    LoadReply;
10
11   port pstruct {
12     covariant message munion {
13       LoadRequest       Load;
14       Store             Store;
15     }                   MemIn;
16
17     contravariant message
18       LoadReply         MemOut;
19   }                     MemIO;
20
21   unit {
22     input MemIO         MemIO;
23   }                     Memory;
24 } Memory;
25
26 namespace
27 {
28   unit {
29     output MemIO        MemOI;
30   }                     CPU;
31   unit {
32     input MemIO         MemIO;
33     output MemIO        MemOI;
34   }                     Cache;
35 } CPU;
```

Program 35 A Simple Computer System

```
1  channel fifopipe<1,1,15,1> FIFO1x16;
2
3  unit
4  {
5    instance ::CPU::CPU CPU;
6    instance ::CPU::Cache Cache;
7    instance ::Memory::Memory Memory;
8
9    channel FIFO1x16
10     Chan1 { CPU.MemOI -> Cache.MemIO },
11     Chan2 { Cache.MemOI -> Memory.MemIO
           };
12 } System;
```

other three units and connects them as shown in Figure 27. The program contains three instance declarations, all of which use fully qualified static identifiers for the units which are being instantiated. This was done in order to ensure that they have been correctly named despite the fact that we have not shown the placement of the declaration for `System` in the namespace hierarchy.

Figure 27 CPU and Memory



`System` also contains two channel instances, which will create four channels, connecting the appropriate port structures of the unit instances. The channel declarations inside of `System` start with the `channel` keyword, followed by a channel model, in this case `FIFO1x16`, followed by the name of the channel and a pair of dynamic identifiers for the ports it connects.

The last feature of note in this program snippet is the channel model declaration at the top. This declares a channel model named `FIFO1x16`, which we reference two times inside the `System` unit declaration. The model is of a 16 deep, 1 bit wide FIFO, with a 1 cycle forward and backwards latency, as outlined in section (see Section 3.3).

7.6 BlinkyExample

The example presented in Program 36 is meant to be the simplest, complete RDL example possible. It includes a single unit (), a platform () and a mapping of one to the other. The unit itself has a Verilog implementation in `Blinky.v` included in the RDLC2 downloads which does nothing more than rotate a single one through a shift register of parameterized with, intended to drive a row of LEDs.

This example adds little that the counter example which does not explain more completely, except that it removes all possible extraneous details.

7.7 Conclusion

In this section we have presented several example RDL descriptions, including two which are complete enough to be mapped and tested in FPGAs or HDL simulation. These examples, in addition to being compiler test cases (see Section 9.6), are meant as illustrations of the basic language features as they apply to real systems. This section is both a reference for new RDL users, and a showcase of how RDL applies to real, if small and simple, systems.

Program 36 BlinkyExample.rdl

```
unit <width = 8, delaybits = 2> {
  plugin "Verilog"
    "External"<"Output", $width> LED;
} Blinky;

namespace {
  platform {
    language "Verilog";
    plugin "ModelSimEngine"
      Engine;
    plugin "ModuleLibrary"
      <"ModuleLibrary.xml">
      Library;
    plugin "ModelSim" Launch;
  } ModelSim;
} Platforms;

namespace {
  map {
    unit ::Blinky Unit;
    platform
      ::Platforms::ModelSim
      Platform;
  } ModelSim;
} Maps;
```

Chapter 8

RDLC Toolflow

RDLC takes a target system, a host system and a mapping between them, all specified in RDL, and produces a simulation or emulation of the target system implemented in a source code forest tailored to the host in question, a process called mapping and described in Section 8.4. In the previous sections we have covered both of the models (see Section 2.4) which underly RDL and the language itself (see Sections 5 and 6), and gave a number of examples (see Section 7) of RDL. The key to translating RDL into a working design is of course RDLC, which is primarily responsible for parameterization, including inference, netlist elaboration to handle hierarchically defined units and platforms, plugin invocation and code generation, often through the plugins.

Like most abstraction tools, RDLC is designed to simply generate a forest of source code (a collection of Verilog or Java directory trees so far) which can then be examined by the designer or fed into the next stage of tools (FPGA synthesis or a Java compiler for example). Unit implementations are excluded from this, because RDL is a system and interface description language meant to tie together gateware or software written in more conventional languages, though RDL plugins may also be used, particularly for highly parameterized units. RDLC has been explicitly designed to make this intermediate code as readable as possible, including the passthrough (see Section 9.5) of names from RDL identifiers and even the preservation of structure where prudent *e.g.* the unit hierarchy to Verilog module hierarchy.

We start this section , in Section 8 with a discussion of the overall CAD tool flow for software and hardware designs, of paying particular attention to how RDLC fits in. We spend significantly more time on the two primary RDLC2 commands: map (see Section!8.4) and shell (see Section 8.3).

The compilation model for RDL is complicated by support for cross-platform designs and independent unit development, and most of all cross-abstraction designs spanning hardware and software. In this section we describe the toolflow for an RDL design, including the process of design, mapping and integration with existing hardware and software toolflows. Most importantly this section documents the process of running RDLC2.

Shown in Figure 28 is a simple illustration of the RDLC toolflow, showing the main RDLC2 commands: shell on top and map below. At the top are the steps to create a unit implementation in a particular host level language, Verilog or Java for example. At the bottom are the steps required to actually produce a runnable simulation or emulation, including running the map command to produce the wrapper and link code, followed by the native host toolflow, be it a JVM, an HDL simulator like ModelSim or an FPGA system such as BEE2.

8.1 Running RDLC

RDLC is implemented as a Java program and distributed as a JAR to ensure that it is easy to maintain and can be run nearly anywhere without porting, an important point for a project as diverse as RAMP. Thus the first step to running RDLC is to ensure that a Java Virtual Machine is installed on the system, and that one has downloaded an RDLC distribution from [9]. Current releases of RDLC will require Java 1.5 or newer, and we highly encourage the Sun JVM [12, 3] as we have seen some undocumented incompatibilities on other JVMs. For the remainder of this section, we will assume that the reader has aliased the command name `rdlc2` to something along the lines of `java -jar rdlc2.jar` with the proper Java classpaths set.

There are generally three ways to run RDLC, the most user friendly of which is the GUI shown in Figure 29. Of course, being mostly tool for computer architects and intended to integrate with large complex projects, RDLC has a command line mode as well. The command line and GUI modes provide access to exactly the same set of RDLC commands, described below, with the same options, thanks to

Figure 28 Toolflow

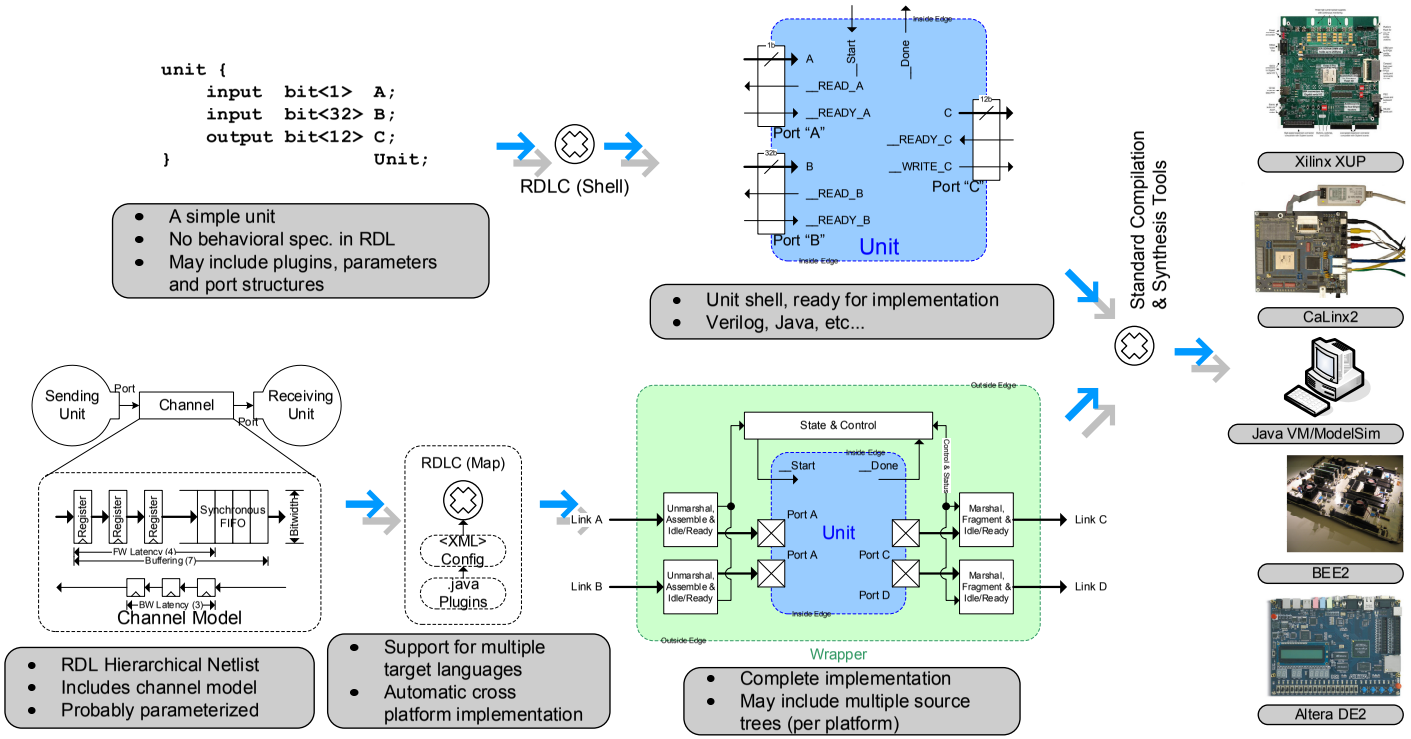
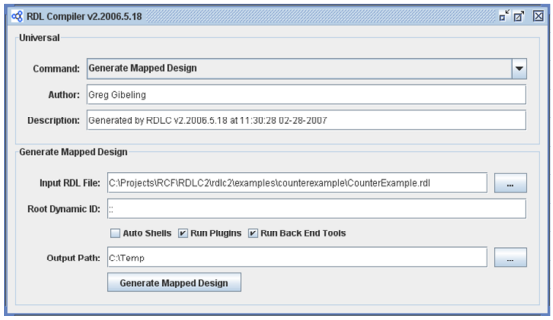


Figure 29 RDLC2 Screen Shot



the infrastructure of the compiler (see Section 9.3).

The third way to run RDLC2 is to invoke the unit test framework, which again has both a GUI and command line mode. The unit tests are included in the compiler distribution as a way to check for JVM portability issues, and to ensure users know they have a fully functional version of RDLC2. We will not discuss the unit tests further (see Section 9.6).

In both GUI and command modes there are several options to RDLC2 regardless of the particular command being run, these are shown in the top half of the GUI in Figure 29. Most notable of course is the command itself, but the author and project description are also available, and used by the code generation routines to mark their output. The GUI features a particularly helpful trick, whereby those options which can be validated (such as the existence of an input file or output path) will be marked red when an invalid value is specified, and white for a valid value.

The command line mode, of course features a `rdlc2 -help` which will dump the entire command line help for all of these commands as well as the version information. It also features a `rdlc2 -license` command which will print out the BSD license. All options and commands should be preceeded by the `-` character. Because several of the options require more complex information, their format is somewhat unique using a quoted, space separated list of values, *e.g.* `-option:`"Value0 Value1 Value2".

8.1.1 Configration

One advanced option not available through the GUI provides for the loading of different or additional RDLC2 XML configuration files. The exact format and nature of these files is described elsewhere (see Section 9.3) suffice to say, that they are necessary only for users writing their own or using third party plugins. In the case of plugin development we recommend obtaining the complete RDLC2 source code, which is freely available under the BSD license from the RAMP Website [9]. For those using third party plugins, we encourage you to bug the creator of the plugin to create their own JAR file with the configuration files and plguin built in to make your life easier.

The `-config` option should appear immediate after `rdlc2` on the command line and before any commands. This option takes two arguments, first a configuration XML file either as an absolute or relative path (`-` denotes the default configuration file), and second a boolean flag indicating whether this config file should replace all current ones or add to them. Thus in order to specify the default configuration file, instead of simply leaving off the `-config` option, one could specify `-config:`"– true" to achieve the same effect. Note that as many `-config` options may be specified as needed to load all the plugins which are desired, though a configuration XML file may load multiple plugins.

8.1.2 Options

There are two options which are universally available for all RDLC commands: `-author:`"authorname " and `-desc:`"description". These allow RDLC2 to embed the name of the author and a description of the project in all generated code. By default these are set to the username under which RDLC2 is run and a string consisting of the RDLC version and time the tool was run respectively.

8.1.3 GUI

Though it is not a command in the compiler sense, one of the most common commands to RDLC2, is `-gui`. This is in fact the default if one were to simply run `rdlc2`, and we present it here only for those users who may wish to combine it with the options listed above. Because the GUI is the default, simply double clicking the RDLC2 JAR on most operating systems will bring up the GUI.

8.1.4 Check

The RDLC2 `-check:`"inputfile dynamic" command is designed to perform a syntax check of an RDL file, in particular the dynamic (see Section 6) named in the arguments, without actually generating any code. This is particularly helpful during development of a larger system when code generation can take several seconds or minutes, and one is stuck on a syntax error. We imagine this command will eventually either disappear or form the basis of an integration between RDLC and an IDE like Eclipse [1] which will allow proper syntax highlighting.

8.1.5 Summary

In this section we have documented the basic RDLC2 options, commands and their arguments. The shell (see Section 8.3) and map (see Section 8.4) commands are documented elsewhere, in a more complete fashion. Interestingly, part of the reason the configuration options must come before commands, is that the configuration files can be used to create additional RDLC2 commands. In truth, though this section documents the toolflow of RDLC2 it really only documents the default toolflow; the configuration options can be used to create others (see Section 9.3).

8.2 Existing Toolflows

RDL is primarily an interface and system description language, relying on behavioral unit implementations being written in other languages. As such a completely mapped RDL design is generally not ready for execution until another compiler or FPGA synthesis tool has been brought to bear. Given the inevitability of a complex tool chain, RDL has been designed to integrate well with these tools, providing the maximum flexibility rather than making concrete expectations about the build environment.

Because the map command (see Section 8.4) produces a tree of source code in whatever language is appropriate for the host system, this code can easily be manually fed to the desired compilation tools. In particular all RDLC1 and RDLC2 language back ends take pains to follow the proper coding style for the language being generated (see Section 10.2). As a direct consequence a mapped RDL design can be easily used with any of the basic toolflows, meaning that RDL can be viewed as a pre-processing step, invoked from *e.g.* a makefile or ANT script.

Though RDLC2 does not assume itself to be in control of the build process, a major detriment we have encountered in some CAD tools [84], it can be used to drive the build process. RDLC2 back end toolflow plugins may be used to completely automate the build process by creating project description files or build files and even invoking the necessary compiler and FPGA programming tools. This allows a design to be mapped, compiled, synthesized, placed-and-routed and programmed onto a board (or run) with a single click from the RDLC2 GUI. This seems a gimicky feature, except that any distributed host, be it software- or hardware-based, will require such automatic design loading tools in order to ensure it is usable by non-experts. RADTools (see Section 11) integration will greatly enhance these capabilities of RDLC in the future (see Section 16.4.3).

We will cover the exact operation of such integration in Section 10.5.

8.3 Shells

In this section we present some example code generated by the RDLC shell command, as well as documentation for the command itself (see Section 8.3.1). Unit shells generated from RDL represent the inside edge interface (see Section 3.2), which is the border between a unit implementation and the remainder of the system. The RDLC shell command translates the unit descriptions, which are interface specifications, in to an implementation language for a programmer to implement.

While it is possible for unit implementations to be automatically generated by RDLC plugins (see Section 10.4.3), it is more common that they will be hand written, drawn from existing code or from outside tools. Hand written units will generally be developed starting from an RDL description, which is the filled in by a designer. The only down side to this approach is that updates to the interface will require the shell to be regenerated, causing the implementation to be lost. We find that keeping the unit implementations in a version control system makes this easy to manage, as the old unit can be replaced and the version control systems merge and diff functionality can then be used to pull the implementation code in to the new shell.

Units drawn from existing code face similar issues to those hand developed, with the only difference that the implementation of such units will likely consist of a simple layer of code which translates the inside edge to whatever interface the legacy code supports. Similarly, unit implementations created by other tools such as Xilinx EDK, will generally involve a simple piece of hand-written interface code in the unit shell which instantiates or calls the external implementation. For more about this subject we would suggest [84] which covers this matter in the context of a real application of RDL: RAMP Blue (see Section 15).

In the remainder of this section we present two unit shells generated by RDLC, and the instructions for how to generate them. Section 8.3.2 shows a simple shell implemented in Verilog by RDLC2 and Section 8.3.3 shows a simple shell implemented in Java by RDLC1.

8.3.1 Shell

The RDLC2 `-shell:`"inputfile dynamic language outputpath" command is designed to produce a shell (see Section 3.2) for the unit named by the static

identifier "dynamic" implemented in a particular language (see Section 8.3). This process is illustrated as the top path in Figure 28.

The resulting file(s) will be put in to the specified outputpath, which should generally be an empty directory. It is common practice for unit implementation files, which the shells are the basis of to appear in the file system in a directory whose path corresponds to the RDL namespace in which the unit was declared, similar to the way file paths and Java package names correspond. The GUI version of this command will include a drop-down box to select among the languages available in the current version of RDLC.

8.3.2 Shell Verilog

In this section we give an example of the inside edge shell Verilog generated by RDLC2 for the CounterExample (see Program 33). The CounterExample.rdl file (see Section 7.4) would be run through RDLC with the command: java -jar rdlc2.jar -shell:''CounterExample.rdl Counter Verilog ./'' resulting in the Verilog shown in Program 37. In truth the generated file would contain a number of comments, including a copy of the BSD license, which we have omitted for brevity. Though this code was generated by RDLC2, the code generated by RDLC1 would be almost identical.

Included in this module is a port declaration for each port (see Figure 5), and a number of control signals (see Figure 12). Notice that in addition to the __Start and __Done signals mentioned elsewhere, there are __Clock and __Reset signals to support platform-wide reset of the the simulation (see Section 4.4.2).

The final six declarations are for local parameters, which in Verilog are constants which can be overridden at instantiation time, for the bitwidths of the various ports, the unit parameters and boolean indicators showing both ports are connected. In a unit with union ports, there would also be local parameters giving names to all the tag values used on the union port. Notice also that the fact that the width of the Count port is set to the width parameter is reflected in the automatically generated Verilog paramters. While the parameter values are given their default values here, the wrapper which instantiates them in a complete implementation will of course override them with the proper values.

With this Verilog shell of the inside edge interface in hand a researcher could fill in the functionality required. Later on, an RDL design incorporating this unit would result in the generation of the wrapper, which is responsible for instantiating this unit. By adding the above file perhaps using the AutoShells option to the RDLC2 map command (see Section 8.4.1), with appropriate functionality, into the synthesis project, a complete RDL hardware simulation design can easily be produced.

In this section we have summarized the basic features of the Verilog inside edge shells, as generated by RDLC.

Program 37 Verilog Shell for Counter.RDL

```
1  module  Counter(__Clock,
2                  __Reset,
3                  __Start,
4                  __Done,
5                  __UpDown_READY,
6                  __UpDown_READ,
7                  UpDown,
8                  __Count_READY,
9                  __Count_WRITE,
10                 Count);
11     parameter   __PARAM_width =     32,
12                 __PARAM_saturate =  1;
13     parameter   __CONNECTED_UpDown = 1,
14                 __WIDTH_UpDown =    1;
15     parameter   __CONNECTED_Count = 1,
16                 __WIDTH_Count =
                       __PARAM_width;
17
18     input       __Clock, __Reset;
19     input       __Start;
20     output      __Done;
21
22     input       __UpDown_READY;
23     output      __UpDown_READ;
24     input  [__WIDTH_UpDown-1:0] UpDown;
25
26     input       __Count_READY;
27     output      __Count_WRITE;
28     output [__WIDTH_Count-1:0] Count;
29 endmodule
```

8.3.3 Shell Java

This section covers only RDLC1, as RDLC2 does not support Java generation, though of course we have a number of future plans (see Section 16.4.1). Shown in Program 38 is the Java inside edge shell generated for the CounterExample (see Program 33). In addition to this one file, the command Java -jar rdlc.jar -shell:''Counter Java'' CounterExample.RDL , will generate a series of support files, for all of the classes and interfaces mentioned in this file.

Java was chosen as the primary software output language of RDLC not because it is well suited to

simulations, but because it is easy to read, easy to implement and has a superset of the features of most other software languages. Thus we believe it is an easy matter to write the output code for other software languages by creating a new set of text renderings for the object oriented software abstraction (see Section 9.4).

What follows is a list of the relevant statements in the Java shell.

Package: There is a package declaration placing this in the root package (`JavaShell`) of the design. In a larger design, the package hierarchy will exactly mirror the RDL namespace hierarchy.

Class Declaration: We are declaring a class named `Counter`, which implements the standard RDL unit interface, `ramp.library.__Unit`. Notice that the unit interface is named as `ramp.library.__Unit` indicating it is the interface `language=Java]!__Unit! in the \lstinline` language=Java]!ramp.library! package, which contains a number of other support classes and interfaces.

Ports: There are declarations for an input and an output port, as represented by the `ramp.library.__Input` and `ramp.library.__Output` interfaces. These interfaces take advantage of generics, a feature of Java 1.5, to use the Java type-checker to ensure that only the appropriate message types can be sent or received on these ports.

Methods: There are empty method implementations for the two methods inherited from the `ramp.library.__Unit` interface, along with JavaDoc references to the original two methods.

Implementing this unit is a matter of adding state, some code in the `public void Reset()` method to initialize it, and some code in the `public boolean Start()` method which will simulate a single target cycle! While in either method, the unit will be able to send and receive messages on the two channels: `UpDown` and `Count`.

In this section we have summarized the basic features of the Java inside edge shells, as generated by the RDLC.

8.4 Mapping

In this section we present some example code generated by the RDLC map command, as well as documentation for the command itself (see Section 8.4.1).

RDL supports declarations for host platforms (*e.g.* an FPGA board or computer with specific input and output devices) and mappings of a top-level unit onto a platform. Platform declarations include the language (*e.g.* Verilog or Java) to generate and the specific facilities available for implementing channels on the host. A mapping from a unit to a platform may also include more detailed mappings to specify the exact implementation of each channel. The back end of the compiler is easily extensible to support new languages, and new implementations.

The RDLC2 map command will produce all of the necessary output to instantiate and connect the various leaf units, in the desired set of host languages. The generated code is designed to be easily readable, and structurally congruent with the original RDL description.

8.4.1 Map

The RDLC2 `-map:`" inputfile dynamic autoshells plugins backends outputpath" command is will generate a forest of directories, corresponding the static structure of the RDL namespaces being processed and containing all of the wrappers (see Section 4.2) and links (see Section 4.3) in a directory named after the map (see Section 6.4) as they are map specific. This process is illustrated as the bottom path in Figure 28. The point of this command is not to generate a specification of a mapping from a target to a host, but to generate the code which implements that mapping and thereby creating a working simulation or emulation. Mapping does not actually require all the units to have implementations to map a design, thought that is, of course a prerequisite for running mapped design. There are a number of arguments to this command documented below.

inputfile: The RDL file in which the desired map can be found. Of course this file may include others (see Section 5.1.3).

dynamic: This is the static identifier of the map to implement. The map must of course contain an instance of the desired top level unit and platform, and may contain any number of submaps as needed.

autoshells: A boolean option (should be `true` or `false`) which control whether shells for units should be automatically included or generated. As stated in Section 8.3.1, unit implementations (completed shells) are expected to be found relative to the input RDL file using a file path generated from the unit's static identifier

Program 38 Java Shell for Counter.RDL

```
1 package JavaShell;
2 /**
3  * Class:  Counter
4  * Desc:   TODO
5  * @author gdgib
6  */
7 class Counter implements ramp.library.__Unit {
8     protected ramp.library.__Input<ramp.messages.Message_1> UpDown;
9     protected ramp.library.__Output<ramp.messages.Message_32> Count;
10
11    /** @see ramp.library.__Unit#Reset() */
12    public void Reset() {
13    }
14
15    /** @see ramp.library.__Unit#Start() */
16    public boolean Start() {
17    }
18 }
```

(name and namespaces). When `autoshells` is `true` these files will be automatically copied to the output if found, or a clean shell will be generated instead.

plugins: A boolean option (should be `true` or `false`) which controls whether "front end" plugins (see Section 6.5.1) will be run during mapping, or ignored.

backends: A boolean option (should be `true` or `false`) which controls whether "back end" plugins (see Section 6.5.2) will be run during mapping, or ignored. This is particular important if there are back end plugin invocations in the platforms which are complicated or expensive, such as FPGA PAR invocations (see Section 10.5).

outputpath: The path in which to create the forest of source code trees which represent the implemented design. The structure and names of these directories will be clear reflections of the namespaces and map being processed. The main directory structure will follow the RDL namespaces in which units are declared, so that their implementations can be reused, but there wlll be a special directory with the name of the map, which contains all of the wrappers, ports and terminals for it. Note that the names of the wrappers and such under the map will match the instance names, the dynamic identifiers, where as the names of all the unit implementations will match the static identifiers. *Generally this should be an empty, non-version controlled directory which is treated purely as intermediate files, meaning that modifications should be to the RDL or original input, not to its output!*

8.4.2 Mapped Verilog

In this section we give an example of the top level Verilog generated by RDLC2 for the CounterExample (see Program 33) when being mapped to the ModelSim HDL simulator. The `CounterExample.rdl` file (see Section 7.4) would be run through RDLC2 with the command: `java -jar rdlc2.jar -map:''CounterExample.rdl ::Maps::ModelSim true true false ./''` resulting in a complex directory tree of Verilog outlined below.

IO A directory containing the dynamics (see Section 6), all of which are unit implementations in this case, which were declared in the `IO` namespace.

Maps A directory containing the dynamics which were declared in the `Maps` namespace.

ModelSim.v The `ModelSim.v` file shown in Program 39. In truth the generated file would contain a number of comments, including a copy of the BSD license, which we have omitted for brevity. Though this code was generated by RDLC2, the code generated by RDLC1 would be almost identical.

ModelSim A directory containing all the dynamic instances declared inside of the `::Maps::ModelSim`, namely the instance of `::CounterExample` named `Unit`.

Unit.v The code implementing the `::CounterExample` unit, placed here because it is a hierarchical unit, and therefore its only implementations are those generated by RDLC2. This is in contrast to the leaf level units declared in the `::IO` namespace.

Unit A directory containing all of the wrapper and port implementations required by `Unit.v`. The wrappers in turn instantiate the unit implementations, and the port implementations are where the link generator (see Section 10.4.1) code ends up. The files in this directory are appropriately named, for example `RDLC__PORT_BooleanInputX_Value.v` is a file generated by RDLC to implement a port, in particular the `Value` port on the instance `BooleanInputX`.

RDLC A subdirectory containing gateware libraries used by RDLC generated code, these are copied from files inside the RDLC2 JAR.

Misc There are a number of miscellaneous Verilog files in the root output directory thanks the use of the `Include` plugin, to copy gateware libraries from the input to the output (see Section 10.3.1).

This section gives a solid introduction to mapping designs to Verilog, and if it is brief that is to avoid documenting things adequately covered elsewhere.

8.4.3 Mapped Java

In this section we give an example of the top level Verilog generated by RDLC1 for the CounterExample (see Program 33) when being mapped to a Java implementation. The `CounterExample.rdl` file (see Section 7.4) would be run through RDLC1 resulting in a complex directory tree of Java outlined below using Java package notation.

Javacounter Contains the Java implementation of units in the `::` namespace, as well as their wrappers. Because RDLC1 does not support parameterization, there was no need to have different wrappers for a unit depending on how it is instantiated. This package includes the `JavaCounter` class, shown in Program 40, which represents the top level map.

Javacounter.io Contains the Java implementation of units in the `::IO` namespace, as well as their wrappers.

ramp Library code, copied from inside the RDLC1 JAR.

ramp.engines Various engines to drive the simulation, these are essentially user level schedulers (see Section 4.4.2).

ramp.library Abstract interfaces for messages, channels, units and the like.

ramp.links Link implementations, including support for the old down channel model (see Section 3.3) which does not account for backward latency.

ramp.messages Implementations of the standard `ramp.library.__Message` interface for different message formats and bitwidths. The need for a separate implementation per-bitwidth stems from Java's inability to have integer valued generic type arguments (C++ can do this), and our desire to use compile time type checking to ensure messages are of the correct type.

Shown in Program 40 is the implementation of the `JavaCounter` class, representing the top level map for this design. Unlike the Verilog implementation of a map shown in Program 39, the software implementation is lengthened, though not particularly complicated, by the flattened unit hierarchy, and sequential nature of software. The important features are lines 6-8 where the unit wrappers are instantiated, which will in turn instantiate the unit shells. Lines 2-4 by contrast all set up the simulation runtime.

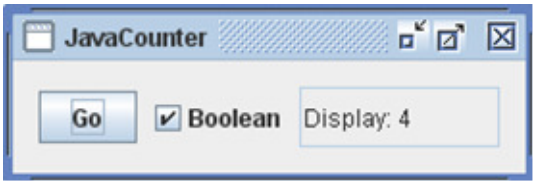
Finally on line 30 of Program 40 is the `public static void main(String[] argv);` function which instantiates this class, resets all the units and starts the simulation. This will result in a small GUI windows, shown in Figure 30 appearing. The checkbox and button together implement the `::IO::BooleanInput` unit, sending a 1bit message every time "Go" is clicked, whose value is determined by the checkbox. In turn the text box reports the value from `::IO::DisplayNum` which will be updated every time the counter changes and sends it a new message. Though internally the complete channel model and RDF semantics are implemented and used a modern JVM, like a modern FPGA, is fast enough to hide this from a casual user.

This section has documented the basic structure, and code of a design mapped to Java. For more complete documentation of the interfaces and classes involved, we recommend the user look at that comments within the code, or the Javadocs which can be generated from them as we have taken pains to ensure their completeness.

Program 39 ModelSim Map for Counter.RDL

```
module Maps_ModelSim( _SW, _BTN);
  input _SW, _BTN;
  wire __Clock, __Reset;

  RDLC__ModelSimEngine __Engine(.__Clock( __Clock), .__Reset( __Reset));
  defparam __Engine.clockfreq = 1000000000;

  Maps_ModelSim_Unit Unit( .__Clock( __Clock), .__Reset( __Reset), ._SW(
              _SW), ._BTN(    _BTN));
endmodule
```

Figure 30 Java Counter



8.5 Conclusion

RDLC is primarily responsible for converting a target system, a host system and a mapping (see Section 8.4) between them in to a forest of source code tailored to the host in question. This includes parameterization and inference, netlist elaboration to handle hierarchically defined units and platforms, plugin invocation and code generation, often through the plugins. RDLC is also responsible for generating unit shells, consisting of the inside edge (see Section 3.2) specification for the unit in question, which an implementor is expected to fill in with the appropriate simulation model of the unit. The compilation model for RDL, as shown in Figure 28, is complicated by support for cross-platform designs and independent unit development, and most of all cross-abstraction designs spanning hardware and software.

In this section we described the process of running RDLC2 through both the GUI and command line, including documenting all of the options, commands and arguments which are available by default. We have described the toolflow for an RDL design, including the process of design, mapping and integration of RDLC with existing toolflows, both as a simple compiler or with push-button build integration.

The integration of RDLC, and hence the tool interfaces and use cases, all derive from the RAMP goals we have laid out (see Section 2.2), particularly the need to integrate with the myriad of existing tools.

Program 40 Java Map for Counter.RDL

```
1public class JavaCounter implements ramp.library.__TopLevel {
2  public ramp.engines.FIFOEngine engine = new ramp.engines.FIFOEngine(true);
3  public ramp.library.__Window mainWindow = new ramp.library.__Window("
       JavaCounter", this);
4  public final ramp.library.__Hashtable<String, ramp.library.__Wrapper>
       instanceTable = new ramp.library.__Hashtable<String, ramp.library.
       __Wrapper>();
5
6  public final io.BooleanInputX_WRAPPER _BaseUnit_BooleanInputX = new io.
       BooleanInputX_WRAPPER("BaseUnit.BooleanInputX");
7  public final CounterX_WRAPPER _BaseUnit_CounterX = new CounterX_WRAPPER("
       BaseUnit.CounterX");
8  public final io.Display7SegX_WRAPPER _BaseUnit_Display7SegX = new io.
       Display7SegX_WRAPPER("BaseUnit.Display7SegX");
9
10 public ramp.library.__Wrapper getWrapper(String instancePath) { return
       instanceTable.get(instancePath); }
11
12 public ramp.library.__External getExternal(String externalName) {
13   if (externalName.equals("mainWindow")) return mainWindow;
14   return null;
15 }
16
17 public void reset() {
18   mainWindow.reset();
19   engine.reset();
20   _BaseUnit_BooleanInputX.reset(engine, this);
21   _BaseUnit_CounterX.reset(engine, this);
22   _BaseUnit_Display7SegX.reset(engine, this);
23 }
24
25 public void start() { mainWindow.start(); engine.start(); }
26 public void stop() { engine.stop(); }
27
28 public JavaCounter() {
29   instanceTable.put("BaseUnit.BooleanInputX", _BaseUnit_BooleanInputX);
30   instanceTable.put("BaseUnit.CounterX", _BaseUnit_CounterX);
31   instanceTable.put("BaseUnit.Display7SegX", _BaseUnit_Display7SegX);
32 }
33
34 public static void main(String[] argv) {
35   JavaCounter topLevel = new JavaCounter();
36   topLevel.reset();
37   topLevel.start();
38 }
39}
```

Chapter 9

RDLC Internals

RDF is designed to be cross-platform, and thus we have found the structure of RDLC and related tools to be vitally important when porting RDL to a new implementation language or platform. For this reason the compiler is highly modular, with very full abstractions and generalizations wherever possible. In this section we delve in to the internals of the two major revisions of RDLC to date, RDLC1 and RDLC2, covering everything from how to run them to the unit tests. This section primarily covers RDLC2, which is significantly more advanced, and has found legitimate, if limited use.

It should be noted that the source code for RDLC1 contains a significant number of comments which can be processed by the Javadoc Tool [5], which should be regarded as the most detailed reference. RDLC2 does not contain such documentation at this time, a consequence of its complexity and the time pressure during its implementation, making this section , not only the first, but the best documentation for RDLC2.

To ease the porting process, and to speed initial development, RDLC is currently written in Java, as Java has a higher functionality-per-line density than many languages and can run on many operating systems. A port of RDLC to C, C++ or a similarly low level language for performance reasons will not be necessary or useful for some time, coupled with the expected increases in JVM implementations, this transition it highly unlikely. Furthermore, as ease of modification and plugin development by a diverse developer community, not to mention portability, are critical RDLC such a port would likely be a step backwards. In a world where researcher time is the key metric, particularly one where FPGA PAR tools with NP-complete problems are involved, a language which allows the clean implementation of improved algorithms is preferable to a faster, but less friendly one.

9.1 Structure

At the surface level, both RDLC1 and RDLC2 are structured around a simple depth-first-search (DFS) traversal of the dynamic instance hierarchy below the declaration specified to the map command (see Section 8.4.1). In both cases while the RDLC core drives the process, it is a language specific plugin (see Section 10.2) which performs the primary DFS to generate code. What is significantly different is the process by which the RDL source code is parsed, the AST is performed, the plugins are loaded and configured and data is passed around inside the compiler.

In short RDLC1 and RDLC2 are designed very differently. RDLC2 is designed to be highly modular, more easily expandable and support more complex language constructs like parameter inference (see Section 5.1.7).

To begin with both RDLC1 and RDLC2 begin with control in a simple loop dedicated to processing the command line options and loading plugins. Both versions support the addition of arbitrary commands to RDLC to share the infrastructure, meaning that even the basic commands like map and shell are built as plugins. Aside from processing the command line the main program's primary duty is load a `Config.xml` file which contains a list of the plugins, in particular the toolflow commands, determine from the command line which one(s) to run and transfer control. The main program also includes functionality which automatically generates the GUI (see Figure 29) and command line help from basic information provided by the command plugins such as argument name and data type.

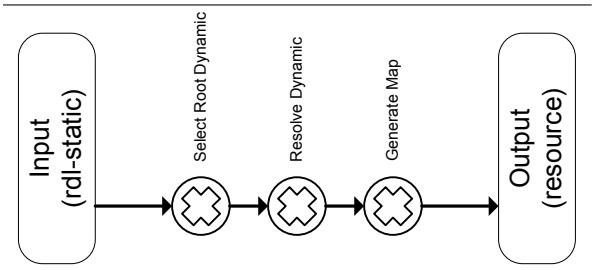
In RDLC1, the main program is further responsible for loading the RDL description specified, and parsing it using JFlex [60] and CUP [54] to create the AST directly. After this point control is transfered to the command plugin which is responsible for whatever other work is required such as code generation or "back end" plugin invocation. The two primary command plugins for RDLC1 are the

Verilog and Java code generators, both of which support the shell and map commands. Internally, both of these command plugins consists of a DFS over the RDL AST, which will in turn generate the applicable code at each step. This structure while nice and simple precluded most of the interesting optimizations and was completely incompatible with parameterization.

In RDLC2, the main program in addition to the above duties, is responsible for constructing toolflows from the XML configuration files (see Section 9.3). In particular the configuration files specify commands, which are build from inputs such as the JFlex- and CUP-based RDL parser, outputs such as Verilog generation and linear chains of transformations. The transformations range from the all-important "ResolveDynamic" to simply filtering the AST to remove everything but the map or unit which has been specified to the map or shell command.

Figure 31 shows the exact chaining of inputs, output and transformations which implement the RDLC map command. First the RDL description is read in, tagged as `rdl-static`, and passed through the `Select Root Dynamic` transformation which finds the root of the dynamic instance hierarchy, in this case the map to be mapped. Second comes the `Resolve Dynamic` transformation, during which the RDL AST is elaborated by static identifier resolution, parameterization and inference and of course type and error checking. Third the map is generated in the `Generate Map` transformation which is responsible to turning a completely elaborated RDL description in to implementation language source code.

Figure 31 RDLC2 Map Command



While we have described the exact steps for the map command, those for the shell command are quite similar.

9.2 Libraries

In order to create RDLC2, as with any compiler, we found it necessary to implement some relatively standard functionality. The choices to implement these library blocks rather than reuse existing ones were usually decided by a combination of the following requirements: BSD license, Java implementation and code simplicity. Nearly all code was ruled out by virtue of its implementation language or a more restrictive license. The remaining candidates, as few as there were, appeared unlikely to save any time in the long run, an estimate which is common among projects of the scale of RDLC.

9.2.1 Error Reporting

A large part of any compiler is ensuring that the input does in fact match the language specification at the lexical, syntactic (see Section 5) and semantic (see Section 6) levels. It is a reasonable estimate that most of the times a compiler is run, it will produce some error with which the designer must contend.

In RDLC1 error reporting was implemented by the simple expedient of printing to `stderr` and quitting in most places. This resulted in unacceptably long development cycles as only the first error could be corrected before compiler had to be re-run. Worse, this meant that if there were two places in the RDL source which together caused an error, *e.g.* a channel connected to two ports of differing type, the error messages were un-helpful to say the least.

To this end RDLC2 includes a more general error reporting mechanism, built around error messages with abstract meanings and payloads being reported to a dataflow DAG of handlers and loggers. This allows error reports to be handled programmatically, including sending them to `stderr` , the GUI or ignoring them altogether in certain contexts. For example without this error reporting mechanism, it would be impossible to imagine the compiler test framework described in Section 9.6. This also helped the reporting of meaningful lexical and syntactic errors by allowing the standardization of source file positions.

The error reporting package `rdlc2.error` contains abstractions of error messages, error handlers & loggers and file positions. In particular there is a Java class `rdlc2.error.ResolveFilter` which will take an abstract error message with a terse string tag and load the appropriate error text from an XML configuration file. This has allowed error messages to be highly standardized, providing designers with much improved feedback, a complaint heard more than once about RDLC1.

9.2.2 IO

Java provides no simple native abstract interface for accessing files stored on disk, in the application's JAR or in memory buffers. The first is necessary for compiler input and output of course, the second to allow the inclusion of pre-built code for inclusion in mapped RDL designs, and the third for testing (see Section 9.6). The `rdlc2.io` package provides just such a set of interfaces, with the additional advantage that it is easily integrated with the tree ADT (see Section 9.2.3) which underlies much of RDLC2.

The abstraction provided by this package actually became vital on some larger designs, as the high file count for RAMP Blue [64] resulted in severe Java runtime errors. In the end, these errors were traced to the fact that RDLC2 was touching a very large number of files, and though closing them the JVM was not garbage collecting file handles before crashing. Adding a simple garbage-collect-and-retry mechanism in the event of failures in this package was quite trivial, whereas in RDLC1 it would have required painful code surgery.

9.2.3 Tree ADT

Compilers, for the most part, are programs which pass around and operate on data represented in ASTs and while Java natively provides a GUI for tree data, it provides no clean ADT for them. As with the other library components in this section, the available alternatives were disappointing, mostly in their refusal to integrate well with the Java collection classes (*e.g.* the `java.util` package).

As such the `rdlc2.tree` package provides a tree ADT, including support for ordering of child nodes, which is important when representing the instances in a unit, and a uniform representation of paths within the tree (for static and dynamic identifiers in RDL). In addition, the class `rdlc2.tree.DFS` implements a DFS Euler Tour traversal of these trees, and is the basis on which all of the compiler transformations, input and output stages are implemented.

In particular each transform, as described in Section 9.1, expects to receive a stream of tree nodes created by an Euler tour. These transformations are generally expected to process the node, perhaps keeping some state, and then pass it to the next transformation. However, they may also decline to process a node (and its children) if unnecessary, *e.g.* in the `Select Root Dynamic` transformation is to prune all but one statement from the RDL source code.

Some transforms may also require the entire tree to be processed before proceeding, or at least some large portion of it. As an example, the `Resolve Dynamic` transform is expected to stitch together all the declarations in an RDL description and elaborate them completely. Given that a unit may be declared later in the source code than it is instantiated, this transform will not be able to pass the instantiating unit on to the next transform until it can resolve the instantiated unit.

Though it seems somewhat complex at first glance, the universal interface within RDLC transformations is a stream of tree nodes. This allows them to be arbitrarily chained, extremely efficient when pruning branches, easily parallelized and best yet, makes them somewhat easier to maintain. In particular, though it is difficult to code the original transformations, this style of programming forces proper abstraction to observed, including careful data structure design. Thus later additions and features, such as some of the major bug fixes months after the original release, remained relatively easy to implement and test.

As a final example, much of the typing and parameterization code does not properly fit within this model, and it is the hardest to read and buggiest code in RDLC. In anything, it is probably that this abstraction should be expanded and better enforced in the future.

9.2.4 XML

In order to bootstrap compiler development, as well as providing an abstraction for configuration files, we implemented a library to create tree ADT representations of XML files. The package `rdlc2.xml` is actual quite simple, being nothing more than a thin wrapper around the `org.xml.sax` SAX parser for input and a serious of print statements for output. This library however proved to be invaluable in testing and configuration, as it could be used to clearly represent compiler trees both for input and output with relying on the JFlex and CUP generated code, which proved buggy at first.

The XML package is also the basis of the ArchSim output language for the Fleet project, as well as the first and simplest output language for dealing with complex mapped designs (see Section 13).

9.3 Config

RDLC2 is highly configurable, allowing the user to specify plugins and even alternate toolflows within the main compiler. In this section we will dissect the configuration files, in particular showing how they can be used to construct source code transformations to create new RDLC2 commands and load simple plugins. Both of these features are used by

more advanced RDL applications (see Sections 13 and 14).

Figure 31 shows the structure of the compiler plugins which comprise the map command (see Section 8.4.1), which as explained above are loaded based on configuration files. Configuration files, as described in this section can be loaded using the `-config` RDLC2 command line option (see Section 8.1.1). There is also an XML schema against which these files must be validated: `rdlc2/Config.xsd` which can be helpful when writing a configuration file.

9.3.1 Toolflows

RDLC2 is structured as a general linear dataflow graph whose elements process tree nodes, presumably pulled from an AST, allowing it to support more than just RDL compilation. There are three kinds of nodes in this graph, inputs, transformations and outputs, which can only be connected to like-typed (same programming language AST) nodes, though a transformation may convert from one language to another. The first task in creating a toolflow is then to declare the language types involved, as shown for the RDLC2 map command in Program 41.

Program 41 RDLC2 Language Configurations

```
1 <language name="rdl-static"/>
2 <language name="rdl-rawdynamic"/>
3 <language name="rdl-dynamic"/>
4 <language name="resource"/>
```

The languages used in the map command listed and explained below. The three variants of RDL represent different stages in the compilation process, which will likely make more sense shortly.

rdl-static: A tree representing the static structure of an RDL description, in particular the namespaces, types and dynamics but not the instantiations.

rdl-rawdynamic: A tree representing the dynamic structure of an RDL description, rooted at a particular unit or map in which the command is interested, but with all the instances intact.

rdl-dynamic: A fully resolved RDL instantiation tree include complete parameter elaboration.

resource: A directory tree, full of file system resources (directories and files), which may be on disk, in memory or part of the RDLC2 JAR (see Section 9.2.2). This "language" is used to represent the output directory and all the files in it.

Having defined the four languages involved in mapping an RDL description, the next step is to define the inputs and outputs. An input, in RDLC2 toolflow paralance, is a Java class capable of sourcing, by some means external to the toolflow, an streaming representation of some tree data. An output, of course is responsible for consuming, and presumably rendering in a useful way, just such a stream of tree data. On line 1 of Program 42 is the configuration declaration for the RDL lexer and parser, which together comprise the `rdlc2.rdl.Input`. Line 2 contains the declaration of the `rdlc2.io.Destination` class which will render tree branches as directories, and tree leaves as files.

Program 42 RDLC2 I/O Configuration

```
1 <input name="rdl-static" class="rdlc2.
    rdl.Input" extension="\gls{rdl}"/>
2 <output name="resource" class="rdlc2.io
    .Destination" extension="/"/>
```

Thus far we have declared four abstract data types representing four different AST "languages" as well as an input which will read RDL files, and an output to render source code directories. Program 43 shows the declaration of the map command itself, starting on line 1, and the the main transformation on line 9.

A command definition, as shown on line 1 of Program 43 includes XML attributes for both the command line name of the command and the GUI text. Furthermore it should contain a `<help/>` element giving the command line help for the command. Finally, it must include an ordered list of `<input/>`, `<transform/>` and `<output/>` elements which define the actual toolflow.

In this example, line 3 declares that the input for the map command comes from an RDL file, lines 4-6 string together the three transforms and line 7 declares that the output goes to a directory tree. Together these 5 lines create the toolflow shown in Figure 31.

Starting on line 9 of Program 43 is the definition of the `Generate Map` transformation. The XML attributes on line 9 include the name of the command as used in configuration files, the input and output languages, which indicate that this transformation will convert a fully resolved dynamic RDL AST in to a series of source code directories, and the Java class implementing the transformation: `rdlc2.TransformMap`. Lines 10-12 define the arguments

Program 43 RDLC2 Command & Transform Configuration

```
1 <command name="map" text="Generate
    Mapped Design">
2   <help>Generate a complete mapping for
      the designed rooted at the
      specified unit, in the specified
      language.</help>
3   <input name="rdl-static"/>
4   <transform name="Select Root Dynamic"
      />
5   <transform name="Resolve Dynamic"/>
6   <transform name="Generate Map"/>
7   <output name="resource"/>
8 </command>
9 <transform name="Generate Map" input="
    rdl-dynamic" output="resource"
    class="rdlc2.TransformMap">
10  <argument name="autoshells" text="
      Auto Shells" type="Boolean"/>
11  <argument name="plugins" text="Run
      Plugins" type="Boolean" default="
      true"/>
12  <argument name="backends" text="Run
      Back End Tools" type="Boolean"/>
13 </transform>
```

to this transformation, which the reader may recognize from the help for the map command (see Section 8.4.1). These elements give the name of the argument, the GUI text displayed for it, the type and a default value. Other argument types include `Boolean`, `String`, `Path` and `Plugin`. The `<argument/>` element may also take a `subtype` attribute to restrict the value further, *e.g.* using a regex to limit the string or path.

RDLC2 has the ability to load multiple configuration files, taking the union of all of them, subject to certain override rules. In particular this will allow a developer to add their own commands in a completely separate configuration file. RDLC2 will then be able to construct the command line help, the GUI and the transform dataflow graph from the configuation information. The flexibility obviates the need for individual developers to create their own GUI and command line processing code, and makes it almost trivial to integrate new toolflows with RDLC2. For examples of this facility see Sections 13 and 14.

9.3.2 Plugin Hosting

Plugins allow RDLC to be expanded and customised in a wide variety of ways (see Section 10). In addition to defining new toolflows and commands, RDLC2 configuration files are how the compiler finds and loads plugins. Configuration declarations such as that shown in Program 44 allow a developer to specify a highly flexible correspondence between plugins and the code which uses, or hosts them[1].

In particular the `<host/>` element on line 1 of Program 44, has a name attribute which specifies the Java class, in this case `rdlc2.TransformMap`, for which we are loading plugins. Line 2 goes on to specify that then the `rdlc2.TransformMap` class requests a plugin with the name `Verilog` it should be told to load an instance of the `rdlc2.hw.verilog.TransformMap` class. This particular example is effectively telling the `Generate Map` transformation defined in Program 43 how to map an RDL target system to `language Verilog;` (see line 7 of program 16).

Program 44 RDLC2 Plugin Hosting Configuration

```
1 <host name="rdlc2.TransformMap">
2   <plugin name="Verilog" class="rdlc2.
      hw.verilog.TransformMap"/>
3 </host>
```

Building on the simple example of Program 44, Program 45 shows some significantly more complex plugin declarations. Program 45 shows a subset of the plugins for the `rdlc2.hw.verilog.TransformMap` and `rdlc2.hw.verilog.TransformShell` classes, which as might be surmised represent the Verilog map and shell transformations. First, it should be noted that these classes are themselves plugins which will be loaded by `rdlc2.TransformMap` and `rdlc2.TransformShell` respectively. Second, this example shows the use of the `rdlc2.plugins.Dummy` plugin, which does nothing.

The `rdlc2.plugins.Dummy` plugin exists in order to allow two a plugin host to avoid an error, and yet not actually perform any work when calling upon that plugin. In Program 45 line 4 specifies that when asked to load the ``MemoryUnit'' (see Section 10.4.3) plugin, the Verilog mapping transformation should load the dummy, whereas line 8 will cause the Verilog shell generator to load the actual `rdlc2.hw.plugins.builders.MemoryUnit` plugin. This plugin is design to generate a partial gateware unit implementation, something which should be part of the unit shell, but which obviously must already have been implemented by the time the unit is mapped. Without the dummy declaration on line 8 however, the map command would fail, with an appropriate error message, as soon as it encountered a line of RDL along the lines of `plugin "`

[1] Not to be confused with an RDF host.

Program 45 RDLC2 Dummy Plugins Configuration

```
1 <host name="rdlc2.hw.verilog.
      TransformMap">
2   <plugin name="BasicEngine" class="
        rdlc2.hw.plugins.engines.
        BasicEngine"/>
3   <plugin name="Include" class="rdlc2.
        plugins.Include"/>
4   <plugin name="MemoryUnit" class="
        rdlc2.plugins.Dummy"/>
5 </host>
6 <host name="rdlc2.hw.verilog.
      TransformShell">
7   <plugin name="Include" class="rdlc2.
        plugins.Dummy"/>
8   <plugin name="MemoryUnit" class="
        rdlc2.hw.plugins.builders.
        MemoryUnit"/>
9 </host>
```

`MemoryUnit" InvokeTheMemoryUnitPlugin;`. A similar, though opposite, situation exists on line 3 and 7.

9.3.3 Error Registry

As suggested in Section 9.2.1 the text of all of the compiler error messages is stored in a separate XML file (`rdlc2/error/MessageRegistry.xml`) in order to ensure that error texts can be helpful and uniform. The message declarations are as shown in Program 46 and are written using Java `printf` format specifier syntax for formatting any arguments. Aside from the error message, line 1 shows the symbolic name of the error used within the source code, and the severity of the error.

Program 46 RDLC2 Error Registry

```
1 <message name="MainConfigLoadError"
    severity="Fatal">Could not load the
     configuration file "%1$s".</
    message>
```

It should be noted that in contrast to the flexible configuration files, the registry of error messages is not extensible at this time, except by modifying this one file. This is easily correctable by moving the error messages to the configuration files, but there has not yet been a reason to do so.

9.4 Organization

As mentioned above, the organization of the RDLC code has had a large effect on the capabilities of RDLC and thus RDL. In this section we, very briefly outline the idea behind the code organization and data structures of the compiler. More complete documentation can be found in the form of Javadocs for RDLC1. The organization described here matches, very closely, the constructs of the relevant languages, meaning there is little left to say.

We have created general, object-oriented software and hardware code generation packages, in order to simplify each language generator in light of the fact that most of them really differ only in the text of the generated output. These packages contain abstraction specific transformation from RDL objects, to either software objects or hardware modules. In RDLC1 these are `rdl.output.oosw` and `rdl.output.hw` respectively, and in RDLC2 `rdlc2.hw`. Though we have only implemented Java and Verilog text generators for these packages, the actual code generation is highly isolated, and *e.g.* converting the Verilog generator to generate VHDL would be trivial.

The point of these intermediate representations is not only to ease the development of new language generators, but to separate the abstraction translation from the code generation, thereby ensuring that the generated code is of a higher quality. As an example, the FPGA synthesis directive generation code is isolated so that adding support for Synplify, XST and Quartus directives took around 20min each, once we found the relevant tool documentation.

There might also be situations where there are two code generators build on the same abstraction, but for different simulation platforms. For example: RDLC will often be used to generate simulations which must be plugged into existing frameworks with their own different standards for communication, naming, typing and timing, but which are implemented in the same language. In this case there might be two separate C output packages which generate different code to make it compatible with existing simulation frameworks.

The input package in RDLC is written as a series of Java classes, one representing each RDL construct (see Sections 5 and 6). For example there are classes for units, channels, messages and ports, as well as the parallel host system constructs. The RDLC front end consists of all of the code required to instantiate and connect all of these classes to create an AST representing the RDL design.

9.5 Mapping, Parameters & Transparency

The code generation back ends, as explained in Section 9.4, were created in part to ensure that the generated code has reasonable textual formatting and style. One of the goals of the RDLC2 map command, especially when parameterization is involved, is to ensure that the code generated is easily readable by a human designer. There is a long tradition of writing domain specific system description tools, like RDL, which generate dense unreadable code, making debugging extremely painful. Even more than this the mapping transformations, such as `rdlc2.hw.verilog.TransformMap` shown in Program 44, were written to be transparent.

As a simple example of this, any unit parameters in RDL will be translated to a suitable, language specific, representation in the output, but with exactly the same name. As a more advanced example, the structure of the directory tree will match the structure of the map, unit and platform instantiation hierarchy. This ensures that even during *e.g.* ModelSim simulation of an RDL design, it is a simple matter to trace the behavior of the system. Examples of generated code can be seen in sections 8.4 and 8.3.

9.6 Testing

As part of the compiler development process we created an automated test suite designed to ensure that with each bug fix or change, things improved rather than breaking old functionality. Compilers in general are large, complex and have a myriad of control paths and special cases, necessitating a concerted effort to ensure they are as bug-free as possible. RDLC is no exception in this regard, and worse than most because of the complete netlist elaboration and parameter inference performed at compile time.

One of the key differences between RDLC1 and RDLC2 is the automation of the compiler tests. In RDLC1 we did maintain a short list of RDL test descriptions, but we did not automate them, and as a result there were far fewer. The tests in RDLC2, described briefly below, provide better corner case coverage as a direct result of the fact that adding a new test case did not entail much work.

To run the unit tests, one must simply run the `rdlc2.test.Main` class, which though not part of the base distribution is freely available. Alternatively, downloading the complete RDLC2 distribution will result in a JAR file which can be invoked using `java -jar rdlc2complete.jar` to bring up a small dialog asking whether to run the main RDLC2 GUI (see Section 8.1) or the unit test GUI.

The unit tester has both a command line, and the GUI shown in Figure 32. The main use of the test GUI is to reduce test success or failure to a single set of easily recognizable icons. It also allows individual test groups to be run and re-run independently, and supports two modes, one in which exceptions are caught and turned in to test failures, and one in which they are not caught for use with an interactive debugger.

Figure 32 RDLC2 Unit Tests

Inside of the `rdlc2.test` package is an XML test description file containing 208 test cases, ranging from simple parser tests, to complete mappings of the main RDL examples (see Section 7). Tests for all corner cases we could think of, and most, if not all compiler errors are included, as are tests designed to catch any compiler errors reported by users. Testing made heavy use of the IO package and the abstract error reporting (see Section 9.2) to include tests which ensure the proper errors are reported.

Of course one annoyance of automated testing using simple file comparison is that line numbers sometimes change thanks to version control. This causes the test suite to report a failed test as shown in Figure 32, as the error message given by RDLC is reported at the wrong source code line. This was deemed a small price to pay for the simplicity of file comparison tests compared to, for example, tests which rely on an external compiler to ensure that generated code works properly.

Some of the unit tests rely on hard coded invocations of various parts of the compiler, but the major ones use the actual RDLC commands. In particular there are several base functionality tests run before even the the test GUI can be displayed, for example a test of the XML library which is used to load the list of tests. A specialized configuration loader was implemented to avoid relying on the XML parsing library (see Section 9.2.4).

RDLC2 was developed using test driven development, as outlined above. This was helpful in that the compiler was always improving, but detrimental in that development was focused on fixing the current set of tests. Next time, we would spend significantly more time designing the compiler, followed by test case construction & development together. Most importantly we would recommend occasionally rewriting code which needs it, despite the perceived short term cost of many failing unit tests.

In the remainder of this section we briefly list the unit tests groups and explain their purpose. Those interested in more detail should refer directly to the XML and test files in `rdlc2.test` as they are relatively simple.

9.6.1 RDL

This section describes the RDL test group, designed primarily to test the language parsing and error checking code (see Section 5).

Messages (Hard Coded) Basic message type declaration (see Section 5.2) tests hard coded to invoke the compiler classes.

Ports (Hard Coded) Basic port type declaration tests hard coded to invoke the compiler classes.

Front End Tests parser & lexer errors, as well as some simple semantic errors.

Advanced Front End Test for complex semantic RDL errors, including bad parameters, array bounds and circular re-declarations.

Messages Complete message type declaration tests based on RDL input.

Ports Complete port type declaration tests based on RDL input.

Unification Basic tests for the parameter inference algorithm.

Units Test the parsing and semantics of units.

Plugins Parsing and the `SetParam` plugin (see Section 10.3.2).

Platforms Parsing and semantics of platforms.

End to End Checks Circularly connected channels and bad channel mappings.

9.6.2 Main

This section describes the main test group, designed to ensure that the configuration loading and command execution loop of RDLC2 are correct. One glaring omission from this group is any kind of GUI test. Given that new users tend to start with the GUI, this is unfortunate, but it does mean that errors will get reported quickly.

Basic Config Loader Test basic configuration file loading (see Section 9.3).

Check Make sure the check command works (see Section 8.1.4).

Include Test the Include plugin (see Section 10.3.1).

Plugin Make sure the plugin invocation interface works (see Section 10.1).

Pretty Print Test an RDL pretty print command which is currently marked incomplete[2].

9.6.3 ArchSim

The tests in this group are based on the ArchSim XML netlisting (see Section 13.5) format which was used on the Fleet project.

Netlist Make sure the ArchSim netlist back end is working properly by mapping designs.

Library Makes sure ArchSim library back end is working properly by generating unit shells.

Commands Run actual commands through RDLC2 to create ArchSim netlists.

9.6.4 Hardware

The hardware test group is designed to test both the complete hardware abstraction package, and the Verilog code generation package which relies upon it. These tests are a mix of hard-coded tests of the code generation packages and full RDLC2 commands.

Verilog Base Test the hardware abstraction using hard coded examples (see Section 10.2.1).

Marshaling A very complex test group which generates marshaling (see Section 4.2.1) logic for a myriad of message types.

Wrapper Generation Make sure the code which maps port structures to flat Verilog ports works.

[2]The actual status of this command's implementation is unclear.

Shells Generate various unit shells (see Section 8.3.2).

Plugins Test all the hardware implementation plugins, memory, FIFO and external in particular (see Section 10).

Maps Map (see Section 8.4.2) some complex designs to hardware and check the results against known working copies.

9.6.5 Misc

There are several miscellaneous test group, some of which are designed for application specific code which is currently semi-integrated with the compiler.

Cross Platform Test mapping the cross platform counter example (see Section 7.4).

Fleet Test the Fleet builder plugins (see Section 13) and examples.

P2 Pretty print some Overlog (see Section 14) code.

9.7 Conclusion

Because of the complexity inherent in RDL thanks to the original RAMP goals (see Section 2.2) we have found the structure of RDLC and related tools to be vitally important when porting RDL to a new implementation language or platform For this reason the compiler is highly modular, with very full abstractions and generalizations wherever possible, allowing us to integrate it our application specific toolflows to produce some very interesting results (see Sections 13 and 14).

In this section we have given rough outlines of the various important areas of RDLC, and particularly RDLC2. The source code for RDLC1 contains a significant number of comments which can be processed by the Javadoc Tool [5], which should be regarded as the most detailed reference. Unfortunately RDLC2 is not yet as well commented, making this section the definitive reference for the high level structure of the compiler.

Chapter 10

RDLC Plugins

In order to allow expansion, and integration of new features, different languages and external tools with RDL, RDLC2 provides a powerful plugin mechanism. RDLC must generate code in a variety of output languages, while providing forward compatibility with new platforms, and therefore new types of links not to mention the need for complex system generation and seam-less toolflow integration. In order to provide this first, a Turing-complete scripting language could be provided within RDL, at a high cost in man hours and complexity. However, only an external language, more focused on software development would provide the code and link generation required, thus prompting the RDL plugin architecture described in this section .

This section is meant to provide basic documentation of the interactions between RDLC2 plugins and the compiler core in Section 10.1. The remaining sections cover purpose and use of the various plugins for output language abstraction, system generation, code generation and external tool integration. Section 9.3 provides a complete description of how plugins are configured and loaded. Section 6.5 provides a complete description of how plugins are invoked from RDL and used.

10.1 Compiler Interface

This section gives a short introduction to the interface which must be supported by RDLC plugins. First and foremmost, the configuration and plugin loading code make no demands of a plugin other than it be implemented as a Java class, allowing the class which hosts it (see Section 9.3) to determine the interface to which the plugin must conform. This said, the most common hosts for the below plugins are the classes which implement RDLC2 map and shell generation and are invoked by the `plugin` keyword or terminal declarations and such. Other than these, the map and shell generators themselves are plugins and are explained in Section 10.2.

RDL plugins are generally given five chances to run arbitrary code at different points in the compilation process.

1. When the parser discovers a plugin invocation in the RDL source code. This is referred to as the point of "existence", and it is generally the first the plugin is invoked, including the call to its constructor.

2. When two plugin invocations must be compared for type equality. For example, this must be implemented by link generators (see Section 10.4.1) as any two terminals connected to a link must be of the same type, and terminal types are determined by plugins (see Section 5.3.2).

3. When two plugin invocations need to be unified as part of the parameter inference (see Section 5.1.7) algorithm. To continue the example from about a completely parameterized terminal and an incompletely parameterized terminal would have be unified if they are connected by a link. This may involve setting all the parameters to be the same, or something else entirely link generator dependent.

4. When the plugin invocation is entirely resolved, *i.e.* when all of its parameters are fully known. Note that plugins are given the chance to report that only some of their parameters need by known, a feature used to good effect in Section 10.3.2. This is referred to as the point of "completion", as the plugin declaration is completely elaborated.

5. Back end toolflow plugins (see Section 10.5) generally hook in to the output code generation flow, allowing them to be notified of each file being generated. This is important in order to generate project or makefiles as needed (see Section 10.5).

More advanced documentation can be found in the form of Javadoc [5] comments embedded in the source code for RDLC2. Of course for the new plugin developer, the plugins discussed below, most of which are implemented in the `rdlc2.plugins` package are good examples to draw on.

10.2 Languages

In this section we introduce the two languages RDLC is capable of generating output in (Verilog and Java) and explain the motivation for our selection, and the internals of the code generation process. We do not give a complete code walk through as the code is dense, and not particularly complicated.

What is vital is that we have created common models of hardware and software. This has allowed us to separate the translation of RDL into to hardware or software from the generation of code in a particular language. Though not all plugins may take advantage of this, choosing instead to output code directly or simply copy it from another file, the core of RDLC relies on these models for portability.

10.2.1 Verilog

Verilog was chosen as the first HDL to be generated by RDLC1 and RDLC2 mostly because it is a kind of least common denominator among HDLs. While there are more advanced HDLs in existence such as SystemVerilog and BlueSpec [56], Verilog is well known and widely used. The choice between VHDL and Verilog was arbitrary and founded mostly on our heavy familiarity with Verilog combined with the knowledge that adding VHDL support would be trivial.

The Verilog language abstraction for RDLC1 is in the `rdlc.hw.verilog` package, whereas for RDLC2 it is in the `rdlc2.hw.verilog` package. These two packages contain similar code, the overall structure of the abstraction not having changed much between compiler revisions. What is notable in the Verilog generation code is that it is truly only a very thin layer of string constants on top of the more powerful hardware abstractions in the `rdlc.hw` and `rdlc2.hw` packages.

These abstractions are centered around a simple view of hardware: netlisting of common components in conjunction with combinational logic expressions and a standard module library. By pushing all behavioral code, particularly state elements, in to the standard library not only does the code generation code get easier, but we are able to sidestep the inter-language differences in state abstractions. Particularly, all the features listed above are shared by most, if not all, HDLs, meaning that this abstraction, with the addition of the proper string constants and some style guidelines could generate any HDL.

Where HDLs like SystemVerilog and BlueSpec shine is in their more advanced language constructs, not shared with Verilog or VHDL. Because the hardware abstraction provides a higher level view of the hardware to generate, code generators for these languages can easily extract more structure if desired. As an example, the hardware model in RDLC2 includes an abstraction of structures meant to be elaborated at compile time. In Verilog, this will translate in to generate statements where possible, whereas in BlueSpec it wouldn't be treated specially, relying on the BlueSpec compiler to perform the needed elaboration without hints.

Furthermore, the standardization provided by this simple model has allowed us to work around significant bugs in the FPGA CAD tools. Tools like XST and ModelSim have both proved themselves troublesome when dealing with more complex hardware. By rewriting parts of our hardware model we have been able, more than once, to change the structure of the generated code to avoid syntax which caused trouble with these tools, without having to track down all the places in the compiler or plugins where code is generated.

As a simple example of the power of this model, adding pin location constraint support for the Altera Quartus [17] tools was a simple matter, requiring changes only to less than 10 lines of Verilog generator code. Without a standardized hardware model, this change could have involved hours of searching through code and countless bugs over months of use. Currently the list of recognized and translated synthesis directives includes:

iostandard: Determines the IO voltage standard.

loc: Determines the FPGA pin location.

invert: A boolean (`true` or `false`) to specify whether the signal should be inverted between the FPGA and the outside world (for active low signals).

Among the other parts of the hardware abstraction are two standard pieces of code which generate circuits for fragmentation & assembly (see Section 4.2.3) and packing & unpacking (see Section 4.2.2). Fragmentation & assembly in particular are highly complex operations involving nested operations over arbitrarily complex message types. Packing & unpacking, though simpler, result in quite complex code involving Verilog `generate` blocks and complicated expressions. We have taken pains to ensure that the hardware generated by this

code is correct, highly efficient and works with a variety of Verilog parsers, some of which have severe bugs we were forced to work around.

10.2.2 Java

Java was chosen as the first software language to be generated by RDLC1 most because it is a kind of least common denominator among object oriented software languages. Furthermore RDLC was written in Java for maintainability and portability reasons, making Java an obvious choice of language to generate. Of course in the future C and C++ generators will be required for performance reasons, as the goal of RAMP is to create fast simulators, not portable ones.

Similar to the abstract hardware model described in Section 10.2.1, the packages `rdlc.sw` and `rdlc2.sw` provide standardized models of object oriented software. These packages in turn are the basis of the Java generation code in `rdlc.sw.java` and `rdlc2.sw.java`, though `rdlc2.sw.java` is incomplete.

These packages provide generic object-oriented software models with the actual Java generation separated out mostly as a bunch of string constants. While not all desirable languages, like C, have an objected oriented abstraction built in, most languages have the ability to support object-oriented concepts, using *e.g.* C `structs` full of function pointers. Thus it should be a relatively painless matter to create the necessary string constants to generate C or C++ from RDL by copying and modifying the Java generation code, and relying on the same generic software model.

10.3 Front End

We use the term "general" or "front end" to describe those plugins (see Section 6.5.1) which are designed to affect the beginning of the RDL compilation process. These include plugins which *e.g.* generate a unit netlist (see Section 13.2.4) or modify some parameters (see Section 10.3.2). In other words these plugins have arbitrary access to RDLC's AST and therefore may modify the current description, as needed.

As described above, these plugins take effect at some combination of the existence and completion points in the compilation process. In particular, most of them will, at the existence point, collect a list of the components of the RDL AST which they need to know more about, and ask the compiler to inform them when the necessary information is known. When all the information is available, they will complete their designated task.

The remainder of this section is a documentation of the operation and use of these plugins, with code examples where appropriate.

10.3.1 Include

The `Include` and `ReInclude` plugins both allow a designer to specify the certain files should be copied from the source code to the generated code when RDLC2 is run. This is particularly useful for including unit implementations or standard libraries on which they rely, without an external build step. Program 47 shows several example plugin invocations which we will explain below, all of which follow standard invocation syntax. Line 1 is taken from the RDLC2 counter example (see Section 7.4), whereas lines 2-3 are taken from the source code to RAMP Blue (see Section 15).

Program 47 Include Invocations

```
1 plugin "Verilog" "Include"<"", "
    ButtonParse.v"> IncludeButtonParse
    ;
2 plugin "Verilog" "ReInclude"<"",
     "Infrastructure/Interchip">
    Interchip;
3 plugin "Verilog" "ReInclude"<"
    proc_utils_v1_00_a", "$XILINX_EDK/
    hw/XilinxProcessorIPLib/pcores/
    proc_utils_v1_00_a/hdl/vhdl/
    conv_funs_pkg.vhd", "PCores/
    proc_utils_v1_00_a/
    proc_utils_v1_00_a_conv_funs_pkg.
    vhd", "(?i)\\bconv_funs_pkg\\b",
          "
    proc_utils_v1_00_a_conv_funs_pkg">

    proc_utils_v1_00_a_conv_funs_pkg;
```

Aside from simply invoking the include plugins there are several arguments.

ReInclude: Though not truly an argument, there are two names for the behavior of the `rdlc2.plugins.Include` class. `Include` will print a warning, whereas `ReInclude` will not. This difference exists to support designs in which one unit is instantiated many times, wherein `ReInclude` should be used to reduce spurious warnings.

PluginRegex: The first argument to the `Include` plugins is a regular expression which all back end tool plugins (see Section 10.5) will use. In particular a back end tool plugin will ignore a file, if its invocation name doesn't match

this regular expression. Thus on line 3 of Program 47, the included file will only be noticed by back end tool plugins declared with the invocation name `proc_utils_v1_00_a`, which in this example happens to be an invocation of the `ISELibrary` plugin described below. As this plugin, like all of RDLC2, is written in Java, it uses Java regular expression syntax.

Source: A relative, or absolute path to the file (line 1) or directory (line 2) to copy in to the RDLC2 output directory. Of course directory copies are entirely recursive, and as shown on line 3, environment variables may be used if prepended by a dollar sign.

Destination: The path, relative to the output directory, in which to put the coped file or directory. This may be omitted, as on lines 1-2, if the relative path should not change between input and output, a common and recommended case.

Replacements: The first 3 arguments, of which the third is optional, may be followed by as many additional arguments as necessary arrange as pairs of regular expression searches and replacements. The replacement on line 3, is designed to avoid naming conflicts by replacing anything which matches "(?i)\\bconv_funs_pkg\\b" with "proc_utils_v1_00_a_conv_funs_pkg". Again, please refer to the Java regular expression documentation for more information.

10.3.2 Parameters

There are two simply plugins designed to allow more flexible parameterization than the per-instantiation functionality provided by RDL2. `SetParam` is designed to support per-platform parameters, and `ResetParam` to support per-array-element parameters. Both of these plugins take two arguments, at least the first which should be a parameter, and set them equal to each other allowing parameter inference to finish the job. However `SetParam` will silently fail if the parameter already has a value, whereas `ResetParam` will force the parameter to a new value, severing its connection to any old values if its already set.

Program 48 shows several examples of the `SetParam` plugin taken from the `CounterExample` unit in the counter example (see Section 7.4). These invocations are designed to be mutually exclusive, with each one running only on a particular platform (see Section 6.5.2). In essence this Program 48 forms a platform dependent case statement, setting the counter width as appropriate for each platform. The fact that the `SetParam` plugin will silently fail if the parameter already has a value would allow us to add the line `plugin "SetParam"<$width, 8> DefaultWidth;` to the end, to give `width` the default of value `8` if no other plugin had set it already.

Program 48 SetParam Invocations

```
1 plugin Platforms::ModelSim "SetParam"<
    $width, 32> ModelSimWidth;
2 plugin Platforms::XUP "SetParam"<$width
    , 4> XUPWidth;
3 plugin Platforms::CaLinx2 "SetParam"<
    $width, 32> CaLinx2Width;
4 plugin Platforms::S3 "SetParam"<$width,
     16> S3Width;
5 plugin Platforms::DE2 "SetParam"<$width
    , 9> DE2Width;
```

The `ResetParam` plugin, though it does roughly the same thing as `SetParam` is applicable to a different set of circumstances. In particular `ResetParam` will sever any equality connection used by the parameter inference algorithm to set a parameter, and force it to a new value.

As shown in Program 49 this is useful as a workaround to the lack of language level support for differing parameters within a unit instance array (see Section 6.2.3). This example shows a few of the plugin invocations used in RAMP Blue (see Section 15) to give each processor a uniqe identifier string. In particular while unit arrays normally imply that each element is identical, including the parameters, these invocations change the `instance_tag` parameter on two of the array elements independently.

Program 49 ResetParam Invocations

```
1 plugin "Verilog" "ResetParam"<
    Processors[0].$instance_tag, "
    microblaze_0"> ProcInstance0_0;
2 plugin "Verilog" "ResetParam"<
    Processors[1].$instance_tag, "
    microblaze_1"> ProcInstance1_0;
```

10.3.3 Generators

Because front end plugins have access to the entire RDL AST they can actually rewrite the entire design if desired. In addition to the simpler examples and RAMP designs, we have to date built two non-RAMP designs on top of RDL, using not as the system description language but as an intermediate

step. In both cases the flexibility and generality of the RDLC2 plugin system is what made our efforts possible.

In particular, we have implemented FPGA versions of the Fleet architecture (see Section 13) and the P2 project (see Section 14) on top of RDL in the process writing some very powerful hardware builder plugins while also adding new RDLC2 commands (see Section 9.3). For hardware builder plugins, the sky is truly the limit; they provide a clean way to programmatically generate RDL descriptions that is integrated with the compiler.

10.4 Back End

In contrast to "front end" plugins which allow for RDL generation, "back end" plugins (see Section 6.5.2) allow for arbitrary code to specify particular implementation details. In particular, while the most valuable units from the research perspective will be portable across platforms by design, library units representing I/O blocks at the target level are inherently not portable.

Back end plugins consist of output code generators of all kinds, and plugins with integrate RDL with existing toolflows. The former are generally invoked by the language plugin (see Section 10.2) and their interface is language specific. The latter generally hook in to the code generation process (see Section 9) and insert themselves as extra steps necessary for code generation.

10.4.1 Link Plugins

Aside from generating wrappers (see Section 4.2) the primary job of RDLC is to generate the links necessary to implement the required channels (see Section 4.3). Link generator plugins are invoked by the language specific code generator, upon request through terminal declarations (see Section 5.3.2), or through platform level default links. Per-platform default links are specified as shown in Program 50 taken from the counter example (see Section 7.4), by specifying a plugin invocation with the name `DefaultLink`. Note that default links like this take effect only within a platform.

Program 50 Default Link

```
1 plugin "RegisterLink" DefaultLink;
```

Whether invoked through the `DefaultLink` or because a channel was mapped to a particular link and terminal, a link plugins job is to generate two pieces of code (see Figure 16) one for each end of the link which are referred to as the "ports" in the generated code (see Section 8.4.2). The compiler is responsible for making any arbitrary wiring or object level connections between these two pieces of code, meaning the link generator is responsible for specifying any external pins or network sockets which will be needed, and for coding data appropriately. Link generators may either make use of the general code generation facilities (see Section 10.2) or they may include library files which are simply pre-written and copied to the output directory. The need for parameterization of course plays a large role in this choice.

To date we have implemented 6 link generators for RDLC2, which a seventh nearing completion. The first four all implement a fixed channel model (see Section 3.3), are meant to exist within one FPGA and are simple examples. Block diagrams for these are shown in Figure 33 along with a list of their timing models in Table 4.

It should be noted that the `TrueWireLink` in particular is quite simple, being just a wire. Furthermore while RDL has no problems with any of these four, the `TrueWireLink` and `BufferedWireLink` are disallowed by RDF (see Section 3.6) because they both have a potential for 0 latency connections.

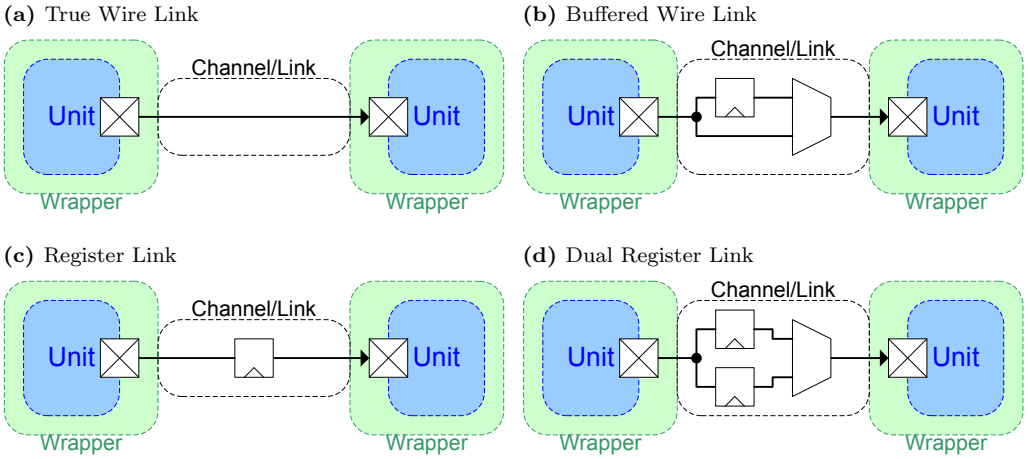
Table 4 Link Timing Models

Link	FW Lat.	Buff.	BW Lat
BufWire	0	1	0
Register	1	1	1
DualRegister	1	2	1
TrueWire	0	0	0

The defining characteristics of these links are how they deal with backpressure. The `TrueWireLink` of course will lose data as it has no buffering, nor cares. The `RegisterLink` by comparison can handle buffering of data, but at the cost of having a 50% usable duty cycle thanks to the backwards latency. The `DualRegisterLink` and `BufferedWireLink` have more complete buffering, allowing a 100% duty cycle, with either 1 or 0 cycle forward latency respectively.

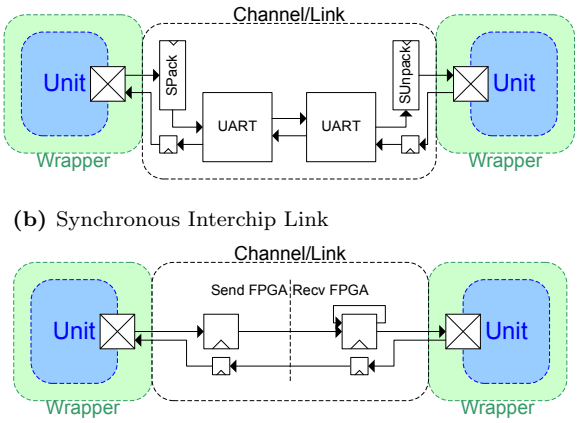
In addition to these four rather simple links, we have implemented two complex links shown in Figure 34. Both of these links are designed for connecting two FPGA platforms. The `UARTLink` will connect any two FPGAs with an RS232 cable, or a similar pair of wires between them, and was chosen as a first demonstration for the ubiquity of serial ports. The `SynchronousInterchipLink` was designed to connect units on different user FPGAs of the BEE2 board, which are connected by wide

Figure 33 Simple Links



high-performance parallel busses.

Figure 34 Complex Links



Shown in Program 51 are example terminal declarations for use with these links, both drawn from the counter example (see Section 7.4). In particular both links, unlike the four simple ones above, expect arguments to specify which FPGA pins various signals are connected, which till translate to synthesis directives through the the language back end (see Section 10.2.1). They do not expect the same pins to be used on two different FPGAs of course.

The arguments for the `UARTLink` plugin are:

TX Pin: The FPGA pin location for transmitting data.

RX Pin: The FPGA pin location for receiving data.

Program 51 Advanced Links

```
1 terminal ("UARTLink"<"AW6", "AV5",
    27000000, 115200>) UARTTerminal;
2 terminal ("SynchronousInterchipLink
    "<34, "C15,L16,M16,J16,K16,H16,G16,
    E16,F16,M18,M17,L17,K17,H17,J17,F17
    ,G17,D16,D17,K18,L18,G18,H18,E17,
    E18,C18,C17,L19,M19,J19,K19,G19,H19
    ,E19">) SynchronousLink;
```

Base Clock: The clock frequency of the board level clock (see Section 10.4.2).

Baud Rate: The desired baud rate for the link, will be approximated as closely as possible given the base clock rate.

The arguments for the `SynchronousInterchipLink` plugin are:

Width: The width of the connection between FPGAs.

Pins: The FPGA pin locations which are connected between FPGAs. These pins are assumed to be bidirectional, as the link must transmit data in one direction and handshaking in the other. Note that both FPGAs must use the same clock frequency, and should probably use the same clock; this is the *synchronous* interchip link after all.

The `SynchronousInterchipLink` in particular is an example of the power and goal of the plugin architecture. It was developed, including reverse engineering of the BEE2 example code on which it is based, within an afternoon at FCRC in San Diego

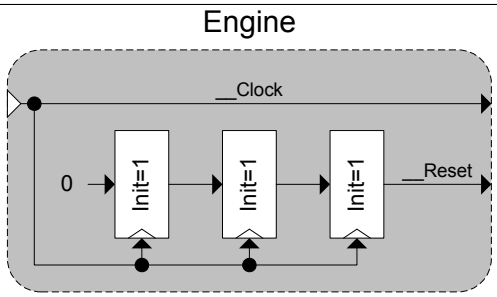
in 2007 at the request of a potential RDL user. More striking is that it while it was not used by requester, it was found to work perfectly upon the first test, performed several weeks later by someone other than the implementor who was not yet an FPGA expert. We hold this as a perfect example of the development model where one person's work can now be leveraged transparently by many.

The final link, currently in development by a researcher working with the author will support the complete timing model within a single FPGA.

10.4.2 Engines

Aside from the wrappers and links, the final component of any RDL simulation is an engine (see Section 4.4.2) to drive the simulation. On a software platform this will consist of a unit execution scheduler, and on a hardware platform this will consist of a module to generate clock and reset signals. The examples in Section 8.4 show the instantiation of engines in both Java and Verilog. Though we have implemented a Java engine for RDLC1, we will discuss only those engines tested and working with RDLC2.

Figure 35 Reset Generating Engine



We have implemented six hardware engines for RDLC2 as listed below, along with a short list of the arguments to be used in their invocations. All of these are responsible simply for generating clock and reset signals from the default clock on the board. Most of them generate the reset signal from "thin air" as it were, using FPGA specific techniques generally based on a small shift chain, as shown in Figure 35, to generate a reset pulse after programming.

Basic: A simple pass-through (without any special buffering) engine for externally generated clock and reset signals. Takes two arguments, specifying the FPGA pin of the clock and reset inputs respectively.

ModelSim: Generates a simulated 100MHz clock signal using a Verilog `always` block. Takes one optional argument specifying the clock frequency to be generated (100000000 would create a 100 MHz clock).

Cyclone2: Generates a reset signal and a properly buffered clock based on simple clock input. Takes one argument, specifying the FPGA pin of the clock and followed by an arbitrary number of pairs of additional synthesis directive names and values to be attached to that clock.

Spartan3: Generates a reset signal and a properly buffered clock based on simple clock input. Takes one argument, specifying the FPGA pin of the clock and followed by an arbitrary number of pairs of additional synthesis directive names and values to be attached to that clock.

Virtex2Pro: Generates a reset signal and a properly buffered clock based on simple clock input. Takes one argument, specifying the FPGA pin of the clock and followed by an arbitrary number of pairs of additional synthesis directive names and values to be attached to that clock.

VirtexE: Generates a reset signal and a properly buffered clock based on simple clock input. Takes one argument, specifying the FPGA pin of the clock and followed by an arbitrary number of pairs of additional synthesis directive names and values to be attached to that clock.

Program 52 shows an example of several engine plugin invocations from the counter example (see Section 7.4), any *one* of which could be added to a platform to specify the engine. The engine for a platform is determined from the plugin invocation, if any, within that platform with the invocation name `Engine`, meaning that there can, of course, only be one engine per platform. Notable on line 2 of Program 52 is the use of an additional pair of arguments, as mentioned above, to tell the `Virtex2Pro` engine that the clock pin on this platform uses the LVCMOS25 IO voltage standard.

Program 52 Engines

```
1 plugin "ModelSimEngine" Engine;
2 plugin "Virtex2ProEngine"<"AJ15", "
    iostandard", "LVCMOS25">     Engine
    ;
3 plugin "VirtexEEngine"<"A20"> Engine;
4 plugin "Spartan3Engine"<"T9"> Engine;
5 plugin "Cyclone2Engine"<"D13"> Engine;
```

10.4.3 Builders

The basic design of RDL assumes that unit implementations will be written in another language, one presumably more suited to the platform to which the unit is mapped. However, there are some units whose basic operation is the same across platform subject to certain parameters. More than this, there is a wide variety of functionality needed for many units, links and other such implementations. To this end we have written a small series of plugins to build these elements. In particular we have written code generators for basic memories and FIFOs, useful for unit and link implementations or any other code generation.

Shown in Program 53 is an example unit implementing a read-only memory as a unit which takes a stream of addresses and responds with a stream of read data. Lines 2-3 declare the address input and data output ports using parameterized widths, to make this unit general. Lines 11-12 then invoke two plugins, on of which is `MemoryUnit`, and the other of which is platform dependent as determined by lines 5-9.

Program 53 Memory Builders

```
1 unit <MemAWidth, MemDWidth, Image,
    MemType> {
2   input bit<$MemAWidth> Address;
3   output bit<$MemDWidth> Data;
4
5   plugin ::1::Platforms::ModelSim "
      SetParam"<$MemType, "
      ModelSimMemory"> SetMemModelSim;
6   plugin ::1::Platforms::XUP "SetParam
      "<$MemType, "Virtex2ProMemory">
      SetMemXUP;
7   plugin ::1::Platforms::S3 "SetParam"<
      $MemType, "Spartan3Memory">
      SetMemS3;
8   plugin ::1::Platforms::CaLinx2 "
      SetParam"<$MemType, "
      VirtexEMemory"> SetMemCaLinx2;
9   plugin "SetParam"<$MemType, "Dummy">
      SetNoMemory;
10
11  plugin "Verilog" $MemType<$MemAWidth,
      $MemDWidth, $Image> Memory;
12  plugin "Verilog" "MemoryUnit"<"Memory
      ", "Address", "Data"> MemoryUnit;
13 } ROM;
```

The `Memory` plugins (`ModelSimMemory`, `Virtex2ProMemory`, `Spartan3Memory` and `VirtexEMemory`) all take the same arguments as shown in line 11.

AWidth: The width of the memory address, which determines the number of entries in the memory (2^{AWidth}).

DWidth: The width of each word within the memory.

Image: A memory image to which the memory will be initialized, through plugin specific means. The memory images should be in hexadecimal ASCII, with one memory word per line. Lines may begin either with data or with a memory address in hexadecimal followed by a colon, specifying that the next word of data should appear at that address and subsequent data at increments of one word each.

The memory builder plugins are particularly useful as they will use the most space-efficient memory structure possible on a given platform. This includes taking advantage of different on-chip memory structures and aspect ratios, and even using technology dependent multiplexing strategies to optimize timing.

The `MemoryUnit` plugin is designed to create a unit from a simple memory, allowing the memory plugins to be generalized memory builders, useful for other purposes. The purpose of this unit is to interface between a simple memory and the inside edge interface (see Section 3.2). This plugin takes either three (for a ROM) or five (for a RAM) arguments, and assumes the read and write interfaces are separate:

Memory Plugin: The invocation name of the memory builder plugin to which the below ports should be connected.

Read Address Port: A string naming the RDL port to get read addresses from.

Read Data Port: A string naming the RDL port to send read addresses to.

Write Address Port: An optional string naming the RDL port to get write addresses from.

Write Data Port: An optional string naming the RDL port to get write data from.

Aside from the memory builders described above, we have created two plugins useful for building FIFOs as shown in Program 54 (an excerpt from Section 7.3).

The FIFO builder, is designed to connect a built memory up to head and tail counters to create a FIFO. The FIFOUnit builder, like the MemoryUnit builder will turn such a FIFO in a complete RDL unit[1].

[1] This is less useful as channels can subsume this functionality, thought it helped in our early development.

Program 54 FIFO Builders

```
plugin Verilog "FIFO" <$Depth, $AWidth>
    FIFO;
plugin Verilog "FIFOUnit" <"FIFO", "
    Memory", "Input", "Output">
    FIFOUnit;
```

The FIFO builder takes two arguments, the FIFO depth and address width, which must satisfy $depth = 2^{AWidth}$. The FIFOUnit builder takes four arguments:

FIFO Plugin: The invocation name of the fifo builder plugin to which the below ports should be connected.

Memory Plugin: The invocation name of the memory builder plugin to which the below ports should be connected.

Input Port: A string naming the RDL port to get input data from.

Output Port: A string naming the RDL port to send output data to.

It should be noted that in the full example code for the FIFO the ports `Input` and `Output` are declared `opaque` (see Section 5.4.5), meaning the FIFO unit has no need to see their structure. Furthermore the FIFO builders are not told the width of the FIFO, instead inferring it from the maximum message width of the ports to which the `FIFOUnit` invocation is told to connect.

10.4.4 External

The final, and inarguably most important code generation plugin is called `External` and will appear in every system. This is the plugin which gives RDL units access to wires, pins or software objects other than channels (see Section 4.4.3). The `External` plugin is the ultimate and only escape from the simulated target system; it is how simulations are connected to the external world.

In software (in RDLC1) the `External` plugin allowed access to a shared pool of objects, to which any unit could get access simply by declaring an external.

In hardware (in both RDLC1 and RDLC2) the `External` plugin allows any unit to connect to a signal which will be present at the top level of the hierarchy, in addition to inputs needed by *e.g.* the engine (see Section 10.4.2). As an example the plugin invocation in Program 55 is drawn from the `::IO::BooleanInput` in the counter example (see Section 7.4). This declaration will invoke the `External` plugin to add a port called `_SW` to the Verilog implementation of this unit. The port is declared to be an input, one bit wide and have two synthesis directives attached to it which happen to be platform dependent.

Program 55 External

```
plugin "Verilog" "External"<"Input", 1,
    "loc", $SwitchLoc, "invert",
    $Invert> _SW;
```

The hardware external plugin accepts the following arguments:

Direction: Can be one of `Input`, `Output` or `InOut`.

Width: The width of the external port.

Key-Value: Any number of key value pairs may be added, including all of the recognized synthesis directives (see Section 10.2.1), or `rename` arguments as shown below.

It should be noted that the `External` plugin is merely a way to access things outside of the RDL target system. In particular it provides no abstraction of I/O, or even indirection at the platform level just a basic "escape" mechanism (see Section 16.4).

This can cause trouble if misunderstood as the "externals" specified this way all share the same top-level name space. This means that two instantiations of a unit with an external will connect *to the same wire*, rather then two separate top level wires. The underlying principle is that an external represents a platform level connection, rather than a kind of "host level port" on the unit. This can be overcome, by specifying "rename" key value pairs after the port direction and width, along with the synthesis directives.

Program 56 External Rename

```
plugin "Verilog" "External"<"Input", 1,
    "rename", "ddr_", "rename", $tag,
    "rename", "_clk"> ddr_clk;
```

Shown in Program 56 is a declaration for an external clock signal connected to a unit which can be isntantiated more than once. By giving the different unit instantiations, the externals will be given different names. In particular the rename arguments values will be concatenated to form a final name for the external like "ddr_0_clk" if `tag` is set to `0`.

10.5 Back-End Tools

RDLC does not, nor will it ever, assume itself to be the only compiler too involved in implementing a design, as it does not include a representation for unit implementations. As such RDLC has been designed to integrate well with existing toolflows (see Section 8), by producing source code for them. Taking this a step further, we have developed several plugins which allow RDLC to drive the compilation, if so desired, resulting in some cases in push-button functionality.

This seemingly trivial feature allows new users to get a design such as the counter example (see Section 7.4) running in a simulation, hardware or software environment in minutes. Even seasoned researchers are often caught off guard when the toolflow for a design is changed, and new ones doubly so. Furthermore, the eventual goal of debugging and management integration (see Section 11.7) can be based on this.

The plugins we have developed to integrate with existing compiler tools are referred to, collectively, as "back end toolflow" plugins because they drive a tool flow behind RDLC. These generally hook in to the code generation process (see Section 9) inserting themselves as extra, though often very simple, steps necessary for code generation. This allows them access to the complete list of resources (files) being generated, from which they common create build or project files to be fed in to the relevant tools. The remainder of this section is detailed documentation for the use of these plugins.

10.5.1 Xilinx

The RAMP project has received extensive support of all kinds from Xilinx, and thus the first set of back end tools are designed to integrate with their FPGA flow. In particular we have developed plugins to run XFlow, create ISE VHDL libraries, run ISE and run Impact. By using *e.g.* XFlow and Impact together we can in fact generate an FPGA bitfile and program a board all from within RDLC2, as shown in Program 57 pulled from the counter example (see Section 7.4).

The arguments to the `XFlow` plugin are the values to appear on the XFlow command line [92], most notable is the second argument which specifies the FPGA, an VirtexE 2000 part in a ff896 package in this example. The arguments to the `Impact` plugin are lines to be written to the Impact batch mode script, most notable are the final two arguments which together assign the bitfile to the third chip on the JTAG string and then program it.

Program 58, pulled from RAMP Blue (see Section 15), shows invocations of the `ISE` plugin, which will generate an ISE project, and the the `ISELibrary` plugin which will generate a VHDL library within it. While the `ISELibrary` plugin invocations are simple, they interact with the `Include` plugin (see Section 10.3.1) in an interesting way, in particular the first argument to `Include` should match the name of the plugin invocation for the library in to which the file should be put.

Each `ISELibrary` plugin invocation will generate a section in the ISE project file generated by the `ISE` plugin. The `ISE` plugin simply creates a ISE project file, putting each argument to the plugin invocation on its own line. These files are in the older NPL format, as we could not find documentation of the never ISE project file formats as of the time the plugin was written. Hopefully this will change in the future.

Program 57 XFlow & Impact Plugins

```
1 plugin "XFlow"<"-p", "xc2vp30-6ff896",
    "-synth", "xst_Verilog.opt", "-
    implement", "balanced.opt", "-
    config", "bitgen.opt"> Compile;
2 plugin "Impact"<"setMode -bs", "
    setCable -p auto", "identify", "
    assignFile -p 3", "program -p 3">
    Impact;
```

Program 58 ISE Plugins

```
1 plugin "ISE"<"virtex2p", "xc2vp70",
    "-7", "ff1704", "Modelsim", "[
    Normal]"> ISEProject;
2 plugin "ISELibrary" proc_utils_v1_00_a;
3 plugin "ISELibrary" proc_common_v1_00_a
    ;
```

Note that we have not mention Xilinx EDK, which is a system level design environment. In particular this is because we have no plugins to allow integration with EDK, though we have done it manually for RAMP Blue. Options for integration include using RDLC to produce an all-Verilog design, which could be packaged into an EDK "pcore." Currently we cannot automate this packaging due to an unforeseen limitation of RDLC2 having to do with discovering the port list of generated Verilog. RDLC can also incorporate black-box units, meaning an EDK design can be packaged as a single unit. Given that an RDL design can be a piece of an EDK design or vice versa, the reader is left to imagine the myriad of ways to nest the output from the two tools.

10.5.2 Altera

In order to help demonstrate the cross-platform nature of RDL, we acquired an Altera DE2 [18] FPGA board and wrote a mapping for some simple RDL designs to it. To make a complete demonstration, we wanted to automate the build process as we had with the Xilinx tools. Thus we implemented plugins for the four main Altera back end tools: QuartusASM, QuartusFit, QuartusMap and QuartusPGM all of which are shown in Program 59.

Program 59 Quartus Plugins

```
1 plugin "QuartusMap"<"--part=
    EP2C35F672C6"> Map;
2 plugin "QuartusFit"<"--part=
    EP2C35F672C6", "--fmax=27MHz"> Fit;
3 plugin "QuartusASM" ASM;
4 plugin "QuartusPGM"<"USB-Blaster", "
    JTAG", "P", "1"> PGM;
```

The plugin arguments in Program 59 are simply the command line arguments to the relevant Quartus executables [17]. In this case, they include the exact FPGA part, the desired clock speed and the JTAG chain position of the FPGA in that order.

10.5.3 Misc

In addition to the series of vendor specific back end toolflow plugins above, we have implemented some simpler ones. In particular we have plugins to support: ModelSim, Java, Javac and Synplify Pro, as well as a simple one to pop-up a GUI dialog box during the mapping process (*e.g.* before programming an FPGA). Examples of all of these can be seen in the complete source code for the counter example (see Section 7.4) which can be downloaded from the RAMP Website [9].

10.6 Command Plugins

Some of the most powerful RDLC2 plugins are not, strictly speaking, RDL related at all. In fact there are several application specific plugins which actually create a completely new toolflow with RDLC2, allowing it to process application specific languages, create RDL and then turn the rest of the work over to the standard compiler. The primary benefit of this integration is how seamless it is, as there needn't ever be an intermediate RDL file, meaning that error line numbers and so forth can come from the original source. Of course this also allows these tools to reuse the RDLC2 code libraries (see Section 9.2), and such.

This has already been discussed in some detail in Section 9.3. For examples of this kind of plugin, please see application writeups in Sections 13 and 14 where we discuss an assembler for modifiable hardware and a higher level hardware compiler.

10.7 Conclusion

Plugins are the mechanism whereby we have separated the generation of code, and integration with existing toolflows from the language. They provide a uniform, flexible set of compiler and languages interfaces and a powerful form of extensibility which has allowed us to cleanly implement a number of large projects without reworking the compiler core. Most importantly, plugins provide a clean way to generate RDL itself allowing the most powerful form of parameterized system generation without the complexity costs of chaining many text-based compiler tools.

In this section we have presented the plugin architecture, and several examples of RDLC2 plugins as well as documenting their use RDL code snippets drawn from working examples.

Chapter 11

RADTools

In this section we present the design of a distributed systems management tool which we have developed using elements of the RDLC3 source code (see Section 16.4.1), in order to ease the task of managing a complex system such as a target or host RAMP proposes to simulate manycore, that is to say large and concurrent, computer architectures using complex FPGA-based platforms. This noble goal is likely to be set back by the simple fact that distributed systems are notoriously difficult to manage let alone set up. If RAMP designs are to be usable by operating system, application or algorithm developers, they must be very easy to manage indeed, as these researchers will have no interest or time to spend learning a complicated new tool.

In this section we present RADTools, system meant to ease the burden of configuration and management of distributed systems. To a large extent we present RADTools in the context of web-services for which it was originally conceived and developed, though we have already added support for more systems such as the BEE2 cluster (see Section 11.7) used by RAMP Blue (see Section 15). We believe RADTools will prove invaluable in light of the RAMP goal (see Section 2.2) of making large scale architectural simulations available to operating system, application and algorithm developers.

11.1 Problem

It is widely accepted that difficulty of managing any distributed system increases super-linearly with the number of distinct component services. For a commercial production service this is bad, but quite possible survivable by the graces of dedicated and skilled management staff.

For a research project this situation is quite untenable. First, managing even a relatively small system needed to perform meaningful experiments can require an unreasonable amount of expensive graduate student, professor and staff time. The cost of sharing such systems among researchers is incredible high, and only palatable in light of the cost of purchasing a duplicate system. Second, and far worse, projects based on design space exploration, like RAMP, and other ideas founded on automated configuration changes are incredibly difficult if not impossible.

The key problem is that even those few systems which provide a good management interface rarely support automation or dynamic configuration changes, and even fewer provide any kind of uniform configuration and control interface. This means that a new administrator must spend hours learning arcane commands, and casual users are bogged down in unproductive management tasks This problem was incredibly evident during the labs for CS294-1 RADS, Fall2006 wherein graduate students with a simple assignment and detailed instructions involving Ruby on Rails web-services, wasted countless hours sorting out basic management tasks for the first time.

As an effort to make management of these systems slightly more tenable, and more important make research using them far easier we have developed this project, dubbed RADTools. RADTools automates the management of service state, from simple startup to crash recovery, and provides a simple, uniform and extensible interface for setting configuration options. The interaction of RADServices (see Section 11.2) is captured by a select few structures over these services, whereas the configuration options are captured by plugins (see Section 11.4.1, see Section 11.7).

11.2 RADServices

By setting up a uniform representation of the components of a distributed system, we can simplify their management significantly. Providing a uniform interface also goes a long way to help automate the process, as it significantly reduces the complexity of designing an automated manager sys-

[0]Excerpts from this section have been presented in prior reports and are thanks in part to Nathan Burkhart and Lilia Gutnik.

tem. This in turn helps to lighten the load on researchers managing such experimental systems as the examples below.

To this end, every service which can be managed by `radtools.services.RADTools` must be represented by an object which implements at minimum the `radtools.services.RADService` interface. This interface includes state management, structure and a uniform abstraction for expanding this interface, both statically and dynamically. In the below sections we describe the state and structure abstractions, however the Javadocs are far better references for any code based on this project.

11.3 State Management

The primary property provided by a `radtools.services.RADService` is "`RADService.State`". This enables some of the most important benefits of RADTools: namely failure management, including both causality and failure of the ability to manage services.

In the current implementation state management has been restricted to positive feedback only. That is to say, we never make assumptions about the state of a service, but instead rely on positive feedback to determine if a service is running, stopped, failed, *etc.*. This is vital, as erroneous assumptions about service state which trigger corrective actions may in fact exacerbate the situation. In the future, when adding assumptions about service state (*e.g.* timeouts and other such tricks) the programmer must be careful to ensure that their assumptions are either acted on in such a way that the situation cannot be exacerbated, or ensure that they first enforce the assumption.

For example if the liveness check for a service times out, the current implementation will mark the service state as unknown. If instead the service is to be reported as failed the check, upon timeout, should first either crash or stop the service, thereby making the assumption of failure true. This will ensure that the pre-conditions for any corrective actions are properly met.

11.3.1 Structure

There are currently three structures over RADServices fully implemented, all of which are based on the RCF tree ADT. These are the composition (see Figure 36), dependency (see Figure 37) and management (see Figure 38) trees.

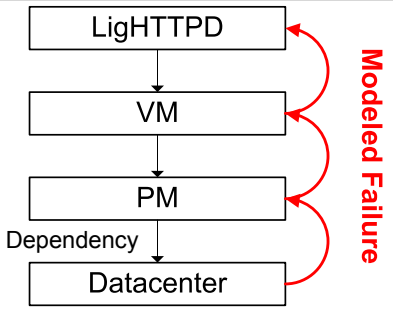
The fourth structure is the communication graph (see Figure 39), which is meant to capture the path of data in a distributed system, in particular requests or RPCs in a web-service. In most web-services a tree model would be appropriate, as all components are RPC-based, however in more general distributed systems, this will often not be the case, and as such we prepared for a graph abstraction.

However the RCF graph ADT was not yet complete and we had little access to path-based analysis tools meaning we had no good means or reason to capture communications paths. As such, this structure is currently unimplemented, though at the time of this writing we are already beginning to remedy this.

The key use of these structures, in particular the fully implemented trees, is to quickly propagate service state changes and manage causality. In the dependency and management trees, failures propagate down as the children of a node are those which depend on it or are managed through it. In contrast in the composition tree failures propagate up, as larger services are built out of smaller ones. This accurately models the fact that, for example, the failure of a physical machine will result in the failure of a virtual machine and that the failure of a database could result in the failure of the entire web-service.

State propagation in the communication graph is slightly more complicated. If it can be reduced to an RPC communication tree, clearly failures propagate up, and this is another kind of dependency tree. However as a general graph the failures must be propagated in the direction of the data flow. Combined with some vertex local information this could generate full failure causality information, something clearly missing from existing distributed systems tools. A simple example of failure propagation is shown in Figure 40.

Figure 40 Failure



Note that while the example in Figure 40 describes and shows the propagation of state changes to `radtools.services.RADService.State#Failed`, other state changes propagate in the opposite direction, as determined by `radtools.services.RADService.State#composition` and

Figure 36 Composition

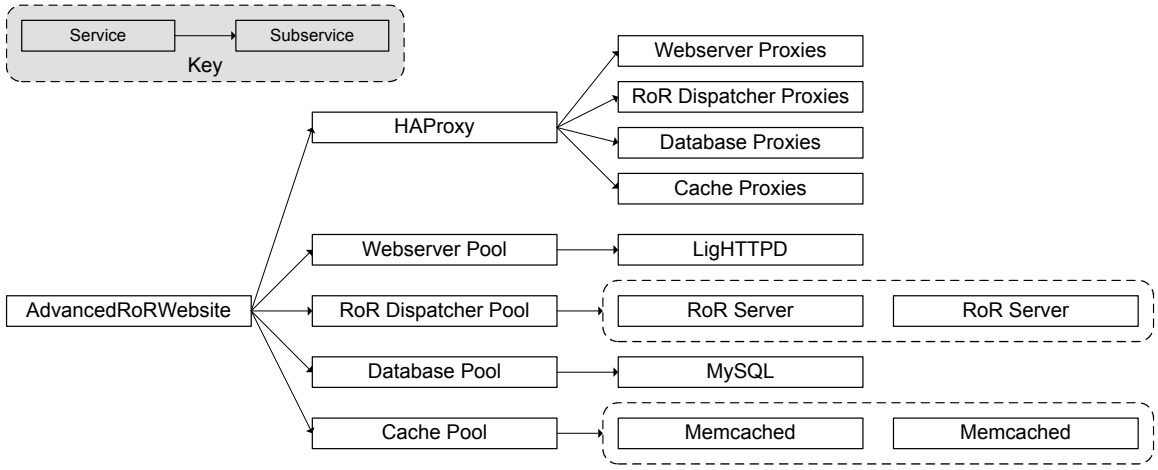


Figure 37 Dependency

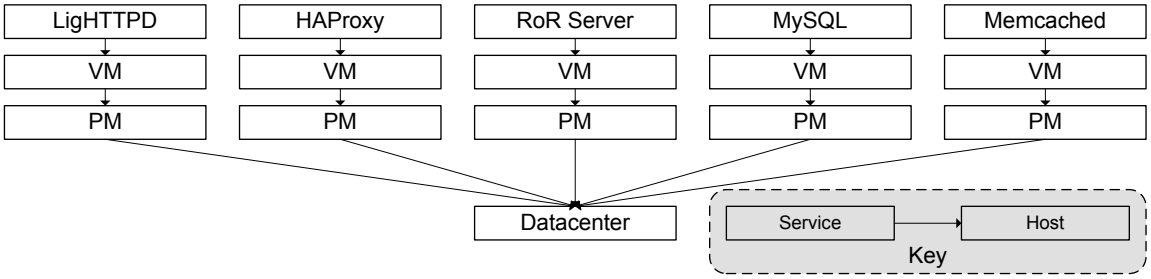


Figure 38 Management

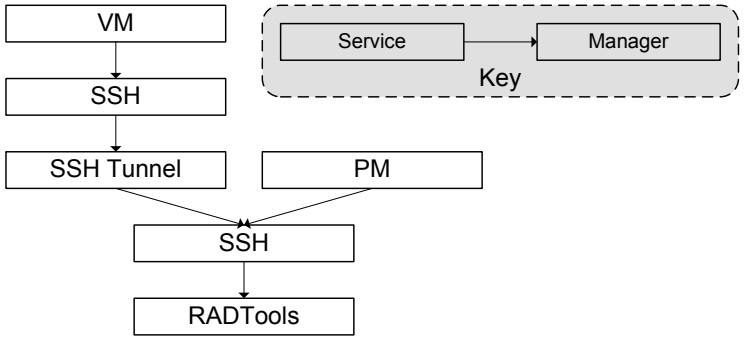
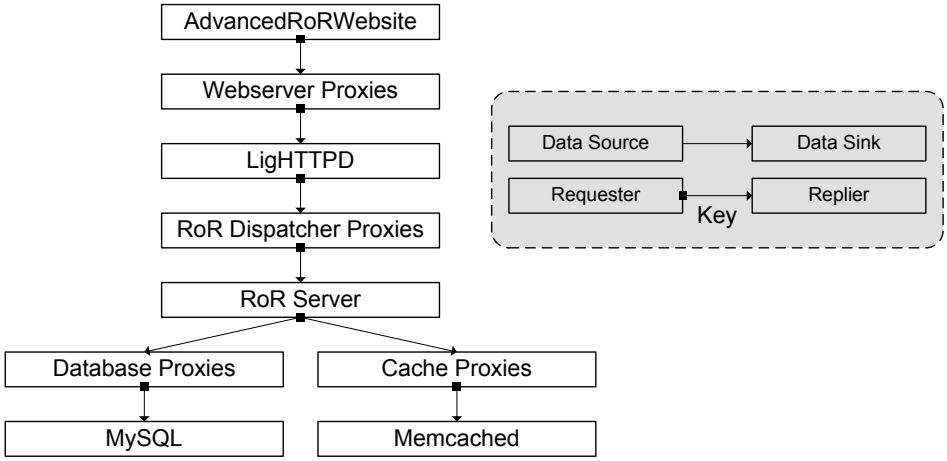


Figure 39 Communication



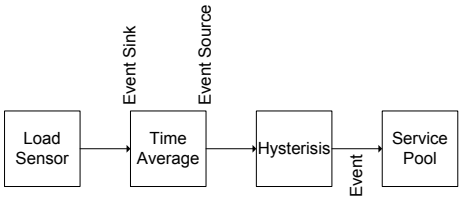
`radtools.services.RADService.State#dependency`, as well as the method `radtools.services.RADService#chain(radtools.services.RADService.Structure, rcf.core.framework.component._dynamic.DynamicPropertyEvent)`. Table 5 lists all of the state changes and structures and shows the chaining mechanism which is used.

The biggest benefit of these structures to the casual user of RADTools is their display in the main window, and the resulting ability to start all the component services of a distributed system in a single click, and the visualization of failures.

11.3.2 Events & Continuations

Events are widely used in RADTools to model causality, and thereby implement policy. For example, most suggested policies for power savings in a datacenter environment are based on starting and stopping servers based on the current service load. Shown below is a possible diagram of the event sources and sinks which could implement a policy like this.

Figure 41 Event Chaining

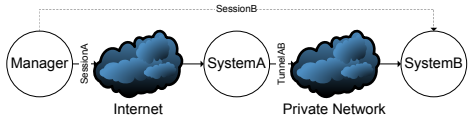


Of course more general examples can be manufactured, but RADTools aims to provide the event framework, rather than implement any specific policy. By creating event handlers which perform the necessary actions, and registering them to receive any useful events a programmer can easily implement a custom policy.

11.3.3 Dynamic Structure

Because management and dependency structures both include RADService representations of sessions, rather than just systems, there is need to support dynamic structures in order to properly abstract the connection and disconnection of these sessions. Figure 42 is an example network, in which the manager must tunnel through *SystemA* in order to connect to *SystemB*. Figures 43 and 44 respectively show the connected and disconnected versions of the various structures (see Section 11.3.1).

Figure 42 Session Tree Net



Because RADTools includes a complete state propogation model, it is easy to capture these changes in system structure as a result of events (see Section 11.3.2).

11.4 Implementation

Our original implementation was geared heavily towards the web-services and experimental setup which was used in the labs for CS294-1, RADS Class Fall2006 [8]. This was driven both by the availability of this setup to test against, and our desire to ease future work on the other projects from

Table 5 Event Chaining

	Start	Running	Stop, Pause	Stopped, Paused	Uninstalled, Restart, Unknown, Failed
Composition	Chain Down	Cond. Up	Chain Down	Cond. Up	Refresh Up
Dependency	Chain Up	Cond. Down	Chain Down	Cond. Up	Refresh Down
Management	Refresh Down				

Figure 43 Connected Session Tree

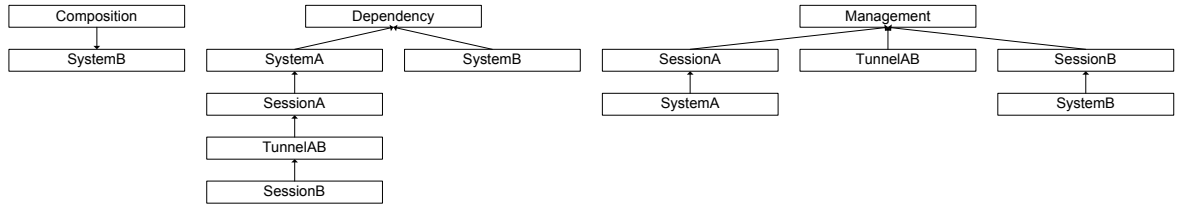
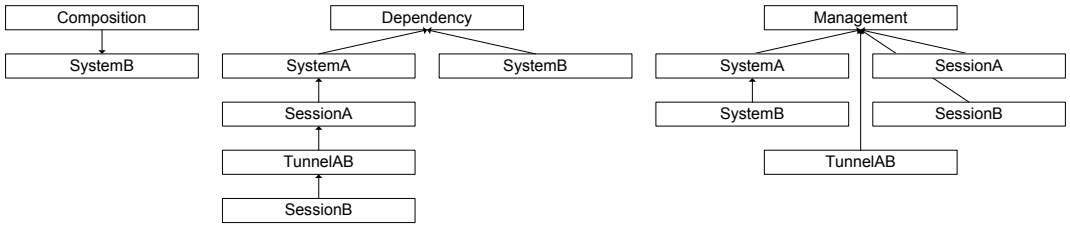


Figure 44 Disconnected Session Tree



that class.

Out choice of the Java language was driven by the availability of the JSCH (see Section 11.4.2) and RCF (see Section 11.4.3) libraries, in addition to the cross-platform compatibility and high level of abstraction provided by Java. In contrast to a collection of shell scripts, this means that RADTools provides a far more useful (and robust) abstraction. In contrast to other main-stream programming languages, this gives us access to a richer set of libraries.

11.4.1 Current Services

We have currently created implementations of the RADService interface for LigHTTPD, HAProxy, Ruby on Rails, MySQL and Memcached. There are also RADServices for VMware, Linux and Fedora Core. In fact the MySQL and Memcached services are currently restricted to Fedora, primarily because that is what we had to test with on the Millennium cluster at U.C. Berkeley.

The linux system RADService includes support for querying nagios, if it is running, and reporting the useful nagios statistics as properties of the Linux system. Currently the support properties are ‘‘`Nagios.CurrentLoad`’’, ‘‘`Nagios.NumUsers`’’, ‘‘`Nagios.NumProcs`’’, ‘‘`Nagios.PercentDiskFree`’’, ‘‘`Nagios.PercentMemUsed`’’. This is a primitive form of service discovery (see Section 11.6.3) and good example of how it might be accomplished.

RADTools is the base service, and represents the RADTools application in the various structures (`radtools.services.RADService#management()` in particular). It includes the code to generate the main window, including tree views of the service structures (Section 11.3.1). Furthermore, as the current implementation of RADTools is focused on the management of large-scale services, the RADTools object includes references to the datacenter, upon which all physical machines are assumed to depend, and the website or service, the ultimate composite service which RADTools is meant to manage.

Finally RADTools includes a queue, which is used to provide scheduling of long running tasks. In addition to allowing a more controlled model of execution, this central queue of tasks is shown in the GUI providing positive feedback to the user. Inter-

nally, this queue is a list of tasks sorted by their scheduled execution time and date. A side effect of this queuing is that duplicate tasks are easily eliminated, greatly increasing the efficiency where very slow services are being managed, an unpleasantly common state of affairs.

11.4.2 JSCH Library

The JSCH library [7] is a fairly simple implementation of the SSH protocol in Java. This was a clear requirement for managing remote Linux-based web-services, and in fact one of the original frustrations that sparked RADTools, was the need to keep around 7-10 SSH sessions open at a time, to manage even a relatively simple web service.

The JSCH library, and its attached compression library can be downloaded from JCraft [6]. While the library itself contains no major documentation, the examples were enough to jump-start our development, despite their quirks.

The largest, and really only, drawback to our use of this library is its design: JSCH includes multi-threaded code without clear documentation of why or when thread safety may be an issue.

11.4.3 RCF

RCF is a set of libraries developed originally for RDLC3, a part of the RAMP project [9]. There are three key pieces to the RCF libraries which are part of this project: data structures, events and components. A large part of the design and development of RCF has been motivated by this project, which as we discuss in Section 11.6.

The transactional data structures are the basis of nearly all of the RADTools code, and provide some vital functionality: the ability of any implementation of `rcf.core.data.collection.Collection` to generate an event in response to a mutation. This is what allows us to write code as in Program 60, which configures `radtools.services.haproxy.HAProxyLinux` to add a new proxy pool, and a new server to that pool.

Program 60 HAProxy Example

```
1 HAProxyLinux.HAProxyPool pool = new
    HAProxyLinux.HAProxyPool(new
    IPv4HostPort(``0.0.0.0:10000''));
2 proxy.pools.add(pool, ``apool'');
3 pool.servers.add(new HAProxyLinux.
    HAProxyServer(new IPv4HostPort(``
    localhost:25''), 22, 3000, 1, 2),
    ``aserver'');
```

The above code is concise, easy to understand and similar to what would appear in the application specific (HAProxy) configuration file, thereby making it easy to learn for those familiar with HAProxy, and easy to automate even for those who are not. However what really makes that three line code snippet interesting is that, because of the transactional data structures, it will actually cause a new HAProxy configuration file to be generated, uploaded over SSH to the server, and HAProxy to be gracefully restarted to use the new configuration. This represents a major step forward in the ability to script the configuration and management of distributed systems.

The second main component of RCF used in RADTools is the event model. As noted above, the transactional data structures rely on the events package, to provide a set of standard interfaces for sourcing, syndicating and sinking events. We omit further discussion as it would merely duplicate Section 11.3.2.

The third and final main component of RCF used by RADTools is the component framework. The component framework provides an abstraction of reflection with extensions for the dynamic addition of operations (methods) and properties (fields) on components (objects). The ability to dynamically add properties (fields) to a component is the basis of our integration with nagios, as seen in `radtools.services.linux.LinuxSystem`.

Furthermore the component framework includes support for generating property change events in response to property changes. This allows the GUI to be kept in sync with the properties, and the configurations to be kept in sync with the GUI all with minimal effort. Please see the GUI package and `rcf.core.framework.component._dynamic.properties.AbstractDynamicProperty#gui(rcf.core.framework.component.DynamicBound.GUIType)` for details about the automatic GUI generation, and property synchronization code.

11.5 Concerns & Obstacles

RADTools has been designed to fill two different, but similar roles. First to allow a person to more easily manage a distributed system, in particular a web service, and second to allow the automation of that management, specifically for research purposes. Second role is easiest to imagine in the case where an automated, (perhaps SML-based, Section 11.6.4) management system is written in Java and linked against RADTools. In this section we strive to document some of our development difficulties in the hope that projects seeking to integrate

with RADTools this way will be able to avoid the pain that we suffered.

11.5.1 JDT Bug

For most any language, and certainly for any large code project an IDE is an indispensable tool, and Eclipse [1] for Java is one of the best. However during the original development of RADTools, there was an Eclipse JDT bug [2] which is relatively uninteresting, except that any user of this code, particularly the RCF libraries, must sometimes work around it.

This bug causes incremental compilation of the RCF libraries to fail after edits of some files, particularly those involving the `rcf.core.data.map` package or the rcf.core.data.collection.Skiplist class. The result will be that random compiler errors will appear in possibly only vaguely related files (including this one, if there is even a link to the skiplist file), often with an error appearing on the first of the line of the file (always a comment in this project). The solution is to perform a clean build using the `Project` → `Clean` menu to fully rebuild the project.

Though this bug has now long been fixed, variants of it have cropped up in nearly every version of Eclipse released since then, always with decreasing impact on our code.

11.5.2 Javadoc Bug

There is a significantly more problematic bug in the Javadoc Tool [5], produced by Sun [11] which makes some Javadocs, the primary source of code documentation, impossible to generate. The release of JDK 1.6 [4] has mitigated, but not entirely fixed this bug.

The problem is in the ability of Javadoc (and perhaps Javac as well), to trace the class hierarchy of certain inner classes, causing it to emit spurious errors and warnings and finally to throw an exception and terminate. We have yet to fully isolate this bug, despite quite some time trying, and therefore have simply omitted that documentation for now. This problem is unfortunate, as Javadocs are quite possibly one of the best code documentation tools in widespread use, however our code in question is quite complicated, and uses complex features added in Java 1.5 [3], so the existence of bugs is not entirely unexpected.

We hope to find a workaround, or a solution soon though in the few years since the original development, the situation has not changed much.

11.5.3 Thread Safety

Both JSCH and Swing use Java threads, without the consent or intervention of the client programmer. The fact that Java threads are ubiquitous, cross-operating system and standardized is wonderful as this makes writing multi-threaded code easy. However both Swing and JSCH sometimes lack appropriate documentation to describe the thread requirements of using them. As a result of this a large part of the development effort on this project was spent debugging threading problems, only two of which could be traced to our own code, or lack of understanding about these libraries.

Please note that for swing the threading reference is the Concurrency in Swing [13] "lesson".

Given how powerful both JSCH and Swing are, we find that even with these problems, using them allowed us to produce a significantly better project in a much shorter time. However the clear lesson here is that any library which introduces threads to a program *must* document how it does so, why it does so and what restrictions the library imposes on the user to enforce thread safety. The one escape clause in this requirement, which we must invoke in places for this project, is that such documentation may only be missing if the threading is provided by a base library which is missing this documentation itself.

An unfortunate consequence of this is that anyone using RADTools as a code base may currently encounter some concurrency bugs. We have not, and we will be more than happy to debug them should they arise, but this is a possible issue.

In general, RADTools follows a simple Swing threading model: long running tasks should be scheduled through `radtools.services.RADTools #schedule(rcf.core.concurrent.schedule.task .TimerTask)`, and GUI operations should be scheduled using `Javax.swing.SwingUtilities`.

As a final note, RCF library provides no thread safety or synchronization, with the exception of the GUI service which will maintain thread safety between a worker thread and the Swing event dispatcher. Any users may also wish to investigate the Class, rcf.core.concurrent.schedule.Runner, rcf.core.util.groups.ImmutableTriple[], rcf.core.base.adapter.Adapter[]) method which can be used to add synchronization to nearly any object or method.

11.6 Future Work

11.6.1 Library Development

A big part of this project was developing the pieces of the RCF library which were needed to implement this project. While the event model and component framework were both well planned out and partially complete, there was a fair amount of work to finish them off.

At the end of this project it has turned out that the code based on these libraries is significantly easier to both write and understand. Furthermore, without them the event and continuation programming of system level policy policy required for SML (Section 11.6.4) would be impossible. As with any library there is still work to be done, everything from using the concurrency support in the `rcf.core.concurrent.primitives` package to simplify the problems outlined in Section 11.5.3, to a more complete AutoGUI in the gui package.

11.6.2 Distributed Implementation

RADTools was designed to provide centralized management of a distributed service, specifically because of the RADLab goal [41] of allowing a single person to design, asses, deploy and operate a large scale web service wherein the single person clearly implies a natural point of centralization.

However going forward with this project it's clear to us that managing a large number of machines from a single point will result in a fairly large load. Currently the management traffic is restricted to simple state updates and occasional configuration uploads, however in the future, access to logs and a larger set of continuous performance data suggests that management of a distributed system, must itself be managed and distributed.

Because of the way the RCF event model and component framework have been designed, it would be a simple matter to extend them to include RMI (Remote Method Invocation), as in JMX, upon which the component model is loosely based. This should enable two major features: first and foremost it would easily allow distribution of the management system without breaking the abstraction in any way, and second it would allow non-java code easy access to the management system, by tapping into the RMI mechanism.

11.6.3 Service Discovery

Currently the structure (the machines in use, and the services they run, but not the configuration of those services) of the system to be managed by RADTools must be hardcoded, for now in `rcf.system.distributed.radtools.Main#inner()`. Given the separate class compilation model of Java this is not an onerous requirement, and yet it would clearly be nice to simplify the process of describing a new system, as this is a painful task and must be completed before RADTools can be used to manage a system.

Obviously adding a simple system description language (such as might be derived from an RDL platform description, see Section 6) would go a long way to decreasing the perceived cost of describing a new system, even if it does not make any real difference, since the Java is quite concise and self-documenting. Even more interesting would be integration with some automatic service discovery system.

There are currently two usage models in mind for RADTools, first, the management of a pre-existing system and second, setting up a new system. Given that RADTools includes the vast majority of the configuration options for the various RADServices it supports, the second model is clearly both preferable and possible, as the initial setup of a distributed web service is often the most painful part. However in both cases, there is information a user should not have to enter. Clearly some things, like the DNS name or IP address of at least one server involved, must be entered. However information like which component services each server has installed, or can run could be discovered by simple inspection of installed programs.

Furthermore path-based analysis could be used both to discover relationships between component services, which could then be reflected by the communication structure . This could be extended to the level of taint tracing through a target or host system.

11.6.4 SML & Plugins

One of the biggest goals of CS294-1, RADS Class Fall 2006 [8] was to bring together Statistical Machine Learning (SML) and Systems graduate students, in the hope of creating hybrid projects. In that spirit one of the main goals of RADTools is to allow a researcher in Statistical Machine Learning, with some Java skill, but no detailed knowledge of web service administration to construct just such a hybrid management system. Goals in this area range from diagnosing problems, and even fixing them, to power and CO_2 conservation.

Our contribution with this project is an abstraction and code base which we hope will remove from future classes and research, the drudgery we felt working with any distributed system and in partic-

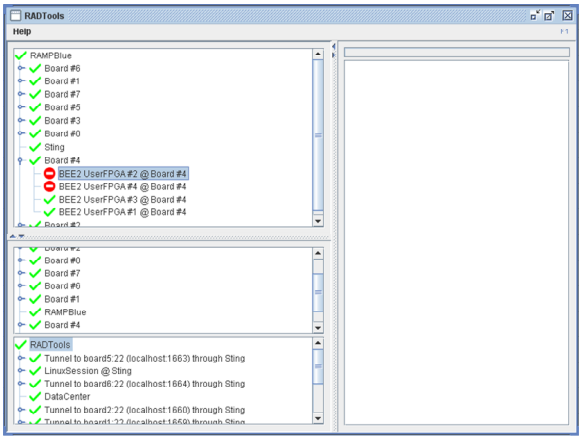
ular Ruby on Rails administration during the class labs. Given the responses of some of our fellow students, we feel we've already gone a long way towards this goal, but time and further projects will tell.

Contributing to this research in a very real way was a major influence on the design of RADTools, primarily in the decision to use the RCF library in order to simplify further coding. For example we use the RCF event model to capture `radtools.services.RADService` state changes, which are propagated by service state proxies through the various RADService structures (most notably management). This event model was specifically designed to be generalizeable to any kinds of events, including periodic performance data gathering, from ``Nagios.CurrentLoad'' to `radtools.services.researchindex_load`. Specifically we have planned that any SML or other "policy" manager should be designed as a series of event sinks which implement DSP or SML algorithms over time series data to produce service control calls, *e.g.* to set a `radtools.services.RADService#radServiceState()`, as shown in Figure 45.

11.7 Integration

Though RADTools is discussed primarily in the context of managing web-services, it has already been modified to help manage BEE2s [25, 37]. Management of a cluster of FPGAs remains a somewhat difficult task as issues like programming, security and simple status notifications are not as easy for hardware. The screen shot in Figure 46 shows RADTools set up to manage the RAMP Blue cluster of 8 boards.

Figure 46 RADTools Screen Shot



RADTools currently has the ability to monitor the programming status of different FPGAs, and to both program and de-program them. This is particularly interesting because RADTools allows this to be done remotely, through multiple levels of SSH tunnels.

The front end machine "Sting" is connected to the Internet, as well as to a local network. The Control FPGAs on each BEE2 have been programming with Linux and are also connected to this local network. Programming a user FPGA then consists of connecting to the Control FPGA through Sting and running a custom script to load a bitfile. RADTools has the ability to load new FPGA programming bitfiles either from the local file system, or automatically use the secure file copy facilities of SSH to upload it to the BEE2 from the management machine running RADTools.

Being able to successfully manage a cluster of FPGA boards like this transparently and with a GUI friendly to new researchers is particularly useful. This is especially true for RAMP where the users of such FPGA systems are likely to be those without any knowledge of FPGA development or CAD tools, and no inclination to learn.

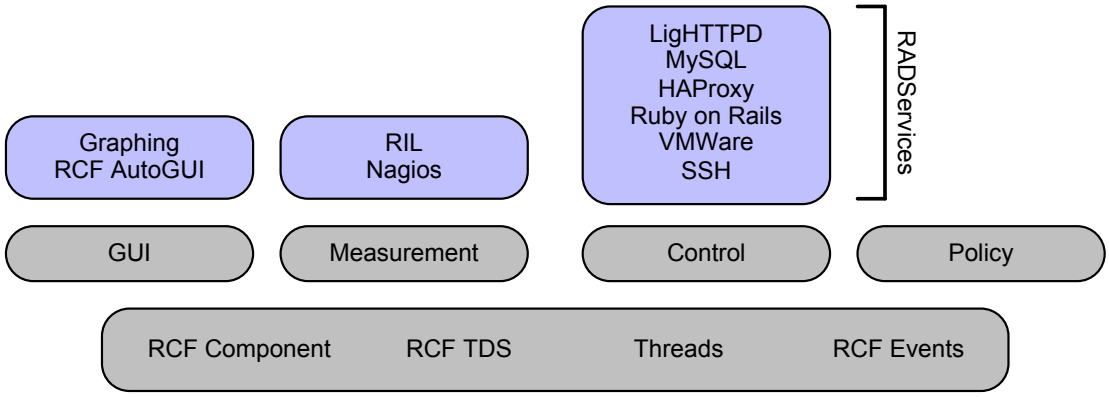
Furthermore, integration with RDL would overcome some of the limitations of RADTools as it stands, allowing the system description, which must currently be hardcoded (see Section 11.6.3) to be drawn instead from the RDL description of the host. It is also conceivable that tools like FPGA timesharing, debugging and soft-core processor management could be added to the mix, thereby solving a number of problems associated with managing an FPGA cluster. Certainly RADTools could be extended to manage HDL simulation tasks as well, thereby forming a unified interface such as required by the application server text.

In particular, because RADTools relies on the RCF library, which is a key part of RDLC3 (see Section 16.4.1) [9], we believe that it will be both easy and very fruitful to adapt RADTools to manage an running RDL host or target system. In particular, RDL provides support for cross-platform system design and emulation, which implies that there are a number of heterogeneous platforms which must all be running components of the same system at once, and working in concert, exactly the scenario RADTools is meant to handle.

11.8 Conclusion

At the end of this project we are now able to, in 5 minutes, configure a complete Ruby on Rails web-application, launch all of the requisite services and benchmark it and have graphs automatically generated (See `radtools.services.researchindex_load`

Figure 45 Framework



`.AdvancedResearchIndexLoadLinux` and ARIL [42]). Even better the RADTools framework allowed us to add management features for a rack of BEE2s within the space of a day. This is no small accomplishment, as the instructions for logging in to and configuring these BEE2s spanned multiple pages, and were only vaguely related to the original web-services goals of RADTools. Thus RADTools is not only instantly useful, but also easily expandable.

In addition to making life easier, this means that more complex, and realistic distributed systems can be easily managed and experimented on, by researchers unable, unwilling or with no interest in understanding the gritty details of their setup. We believe RADTools will prove invaluable in light of the RAMP goal (see Section 2.2) of making large scale architectural simulations available to operating system, application and algorithm developers.

Chapter 12

Mapping

In this section we analyze and design a general algorithm for the problem of mapping RDL units to platforms.

RDL is a hierarchical structural netlisting language for composing message-passing units, designed for specifying FPGA-based gateware simulators for the RAMP project. These designs have a wide range of structures and hundreds to thousands of units, making it desirable to automate the mapping of units to platforms (FPGAs). This section analyzes the problem and presents a complete integer programming formulation, followed by a short description of our IP solver implementation.

12.1 Introduction

As of now, a human is required to specify which units map to which platforms. While RDL makes this specification concise, designing it remains difficult even for highly regular structures. In this section we define the problem, formulate it as an Integer Program and describe a simple IP solver we have developed. In the future we hope to put all of these pieces together to provide a complete tool, allowing a designer to simply specify a collection of units and platforms without designing the mapping by hand.

Section 12.2 defines the problem, including background about related CAD problems in Section 12.2.1, and what makes RDL unique in Section 12.2.2. Section 12.3 briefly describes the standard set of CAD algorithms, including their strengths and weaknesses. Sections 12.4 and 12.5 present our solution and current implementation. We conclude in Section 12.6 and describe the future work in Section 12.7.

[0]Excerpts from this section have been presented in prior reports by the author.

12.2 Problem

RDL is a powerful language for the description of distributed, message-passing systems, especially those which include accurate time accounting for simulation. Originally developed for the description of RAMP multi-core simulators, the language has since found a place in a variety of projects requiring large gateware designs (see Sections 13, 14 and 15).

RDL includes three primary components: the hierarchical unit netlist, the hierarchical platform netlist and a mapping between the two. The basic tool flow for using RDL is simple. An RDL system description is fed through RDLC which produces platform specific source code, *e.g.* Verilog for FPGAs and Java for PCs.

In order to keep the implementation of RDLC simple, the designer must specify the mapping of units on to platforms and channels on to links (see Section 6.4). However, as the RAMP project and other users of RDL, wish to investigate systems with 1000s of units, and 100s of platforms, and a variety of topologies, this will become a burden. Finding an optimal mapping is clearly $NP-hard$, as can be seen from the combinatorial solution space. With design constraints, finding even a reasonable mapping will often be outside the capabilities of a human designer.

Simply stated the problem which we are interested in is the automated partitioning of a communications graph (RDL units) into clusters (RDL platforms), subject to a variety of constraints and goals (see Sections 12.2.3, 12.2.4 and 12.2.5) Given resource limits (*e.g.* memory size limits), and type restrictions (some units can only be implemented in FPGAs) there are clearly infeasible mappings. We therefore need an algorithm which can provide an optimal mapping, against a set of cost-based metrics, while satisfying a range of constraints.

The bad news is that this problem is quite clearly $NP-hard$, and may be infeasible. The good news is that most interesting designs (see Sections 13, 14

and 15) will have been, to some extent, planned by the human engineer. This means that while the problem may be entirely intractable in the worst case, in the common and interesting case it is likely to have a "good" solution by virtue of the foresight of the designer.

In Section 12.2.1 we describe several standard CAD problems all of which are similar to RDL mapping. Following that, in Section 12.2.2 we discuss what makes RDL mapping unique among these CAD problems. Finally in Sections 12.2.3, 12.2.4 and 12.2.5 we detail the constraints which a mapping must satisfy and the goals of an optimal mapping.

12.2.1 Background

In this section we discuss several classes IC CAD problems, and at least briefly mention the algorithms commonly brought to bear.

First, the "mapping" problem commonly refers to "technology mapping" [57]. The input is a graph representation of a digital logic circuit, where nodes represent combinational logic elements, and edges represent wires. The output is a graph which implements the same logic, but whose nodes are technology specific: for example transistors in an ASIC or LUTs in an FPGA. We mention this problem mostly because it shares a name with ours, but it does not share any major features, and so we set this problem aside.

Second, a design which has been "mapped" must generally be "placed". "Placement" is the problem of taking the graph generated from mapping, and finding a two dimensional embedding of the vertices which will minimize the estimated cost of edges. Placement is an $NP-hard$ problem, which consists of embedding optimization against a heuristic estimate of edge cost, often approximated with a distance metric. Notice that the dimensionality of the resulting placement (2D) is a restriction imposed by the fact that ICs are planar.

This leads us directly into the third problem: "routing" [71]. "Routing" is the problem of taking a placed design, a graph whose vertices have been given a two dimensional position, and implementing the graph edges using whatever physical media is available. Routing is complicated by design rules, *e.g.* that routes must fit a grid pattern, and contention caused by a scarcity of routing resources (*e.g.* not many wires which do or can go between places).

Classically placement and routing are heavily intertwined [21], primarily because the overall goal of the two problems is to find a two dimensional embedding of both the vertices and edges of a graph.

The final, and possibly most relevant classic IC CAD problem is "partitioning" [39, 27, 65, 32, 77, 58, 40, 95, 20]. This is the problem of clustering a graph according to some kind of metric, generally with the goal of splitting a large circuit into multiple ICs. Of all the cited papers [39] is perhaps the most classic algorithm. While partitioning, at first, seems to be exactly the RDL mapping problem, there are some important differences outlined in Section 12.2.2 below.

12.2.2 Differences

RDL mapping is superficially identical to IC partitioning, but in this section we give several major differences.

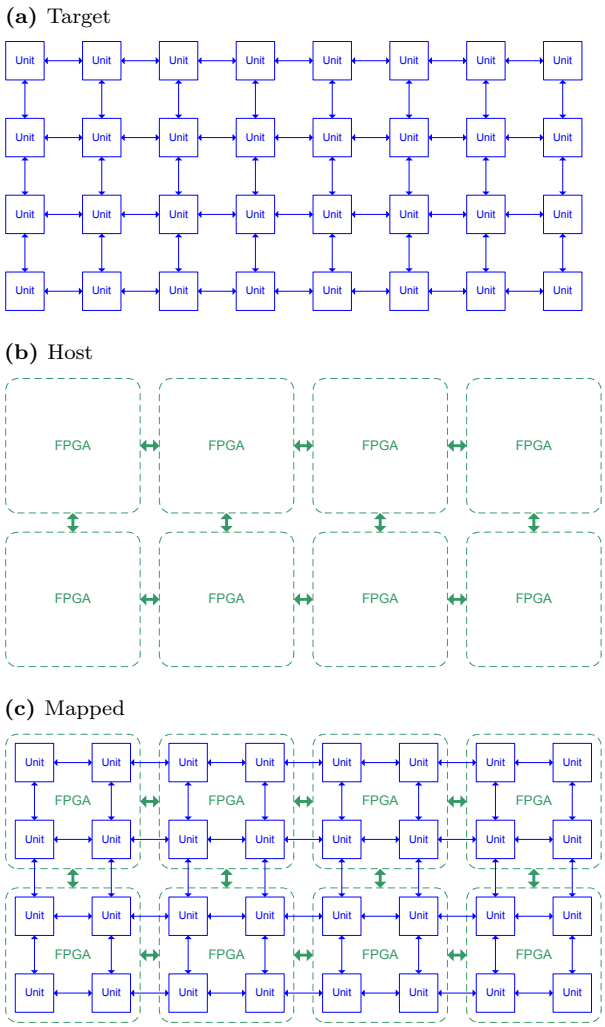
IC partitioning attempts to cluster combinational logic gates into ICs. RDL mapping however attempts to clustering units on to platforms, as shown in Figure 47 The difference here is in scale, RDL units, particularly for RAMP simulators, are approximately 10,000 gates. This implies that algorithms, or heuristics, whose runtime grows too quickly to be useful for IC partitioning may in fact be viable for RDL mapping.

There is a similar scale dichotomy between RDL channel to link mappings and IC partitioning of "nets" or wires. Because an RDL channel is a heavy-weight construct comparable to a distributed FIFO (see Section 3.3), there are necessarily fewer of them in a system than wires in an IC. This reinforces the relaxed runtime requirement mentioned above, and implies that extra care must be taken when mapping RDL channels to links. Because channels can have a timing model, a user specified bitwidth, buffering and latency, the channel to link mapping may incur either resources in the form of extra buffering, or extra host cycles to properly implement the channel timing model. Thus the cost of an RDL mapping depends on the link to which a channel is mapped, whereas all wires are identical in IC partitioning.

Finally, all of the IC CAD problems, whether applied to ASIC or FPGA designs make the assumption that the design will be compiled once and used over a long period of time. In contrast, RDL designs, in particular those for RAMP, may be compiled hundreds of times with slight differences either for debugging or research purposes. Thus any algorithmic solution to the RDL mapping problem must take compilation time into account. See Section 12.2.4 for more information about this particular difference.

Overall the combination of fewer, heavy weight constructs (units, channels, platforms and links) and the need to include compilation time ensures

Figure 47 RDL Mapping



that existing CAD algorithms for partitioning will not suffice for RDL mapping.

12.2.3 Run Time & Resources

The primary goal of RDL is to create cycle-accurate distributed hardware simulators, principally built on FPGA platforms. The obvious goals of the RDL mapping algorithm should be to minimize the size of the resulting simulation, and minimize the number of physical clock cycles which are wasted to simulation overhead.

One of the main differences between RDL and standard IC CAD is the automatic time dilation supported by RDL and the timing models associated with channels. In order to minimize the cost of implementing these timing models, channels should be mapped to links with similar timing. Thus mapping a high performance channel to low performance link will cost simulation time, whereas mapping a low performance channel to a high performance link will cost area for buffering messages. This implies that there is some minimum cost mapping, and justifies our statement that RDL mapping is $NP-hard$. Note that correctness is not affected, as RDLC provides runtime guarantees that proper channel timing is seen by the units.

Constraints in this category include the feasibility constraints on the type of the RDL unit and platform. Clearly a unit with only software implementations cannot be mapped onto an FPGA, and vice versa. [1] Similarly, there may be resource constraints, for example, that a unit which requires access to physical memory must be mapped to a platform which has such memory.

Finally, two units connect by a channel must be mapped to two platforms connected by a link. In this case the channel between the units must be mapped to one of the links between the platforms. RDLC provides no support for multi-hop routing of the messages carried by channels, though it does provide facilities for multiplexing multiple channels on to a single link.

12.2.4 Compilation

Unfortunately the standard FPGA CAD tools upon which most RDL users rely, can take anywhere from 1 minute, to 30 hours. Thus, there is a huge compilation time penalty each additional FPGA-based design. In other words there is a major cost savings for sharing a single design between two platforms.

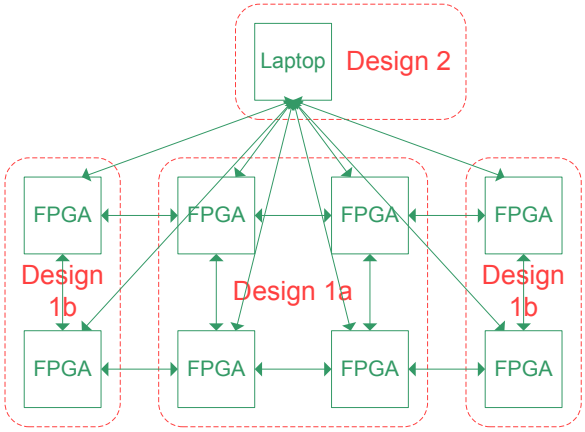
The problem here is to ensure that the post RDLC compilation time scales with $O(1)$ or, perhaps $O(log(n))$ rather than $O(n)$, as the constant factors are very high. While this goal is seemingly very simple, it distinguishes the RDL mapping problem from all IC CAD problems, and significantly complicates this work.

Figure 48, shows an example system consisting of 16 FPGA platforms and a PC connected to them, for *e.g.* bootstrapping or control. Clearly there must be two designs, as an FPGA and a PC cannot instantiate the same kinds of designs even.

This system might have two to sixteen FPGA designs. A single design would be possible if the compiler core, outside the scope of this work, can differentiate the 1a and 1b platforms at load time rather than compile time. Two designs might be needed if the compiler core can only differentiate the 1a and 1b designs at compile time. Sixteen designs might

[1] An HDL simulator may be used to create a software implementation of a hardware design, and a processor may create a hardware implementation of a software design. This duality is out of the scope of the initial work.

Figure 48 Design Minimization Example



be needed if our efforts in this section entirely fail to minimize the number of designs.

12.2.5 Recursive Abstractions

Because RDL is a structural language, there is no reason why a unit could not itself be an RDL system. This kind of recursive use of RDL is actually quite valuable for, *e.g.* debugging infrastructure construction, debugging a simulator and even for the design of some more complex systems.

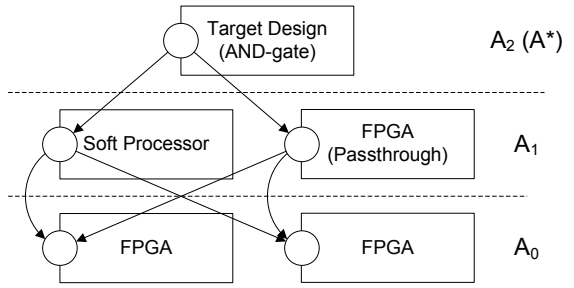
The most powerful form of design recursion, makes target system (units and channels) the host (platforms and links) of another, more abstract, target design (see Section 6.6.2). Using the terminology from [44, 45], this implies that we have three levels of abstraction, $A_0, A_1, (A_2 = A*)$ in increasingly abstract order, each of which is used to implement the one above. Thus we have a third, and far more complex, set of constraints and goals: we would like to be able to solve several levels of the RDL mapping problem at once.

Quality of mapping and the observance of constraints together drive the need for a unified solution. Given that the quality of the mapping is dependent on which $A*$ objects (units) are mapped on to which A_0 objects (platforms), we must clearly solve the mapping problem from end to end at one time.

We might imagine simply omitting design information from the intermediate abstractions $\{A_i | 0 < i < *\}$. Unfortunately, there may be restrictions in these intermediate abstractions (*e.g.* that two A_2 objects belong on the same A_1 object) which must be observed.

Furthermore, the heterogeneity of the objects within a level would make this quite hopeless. Imagine the situation in Figure 49, where we have two A_1 objects: a processor and an FPGA, two

Figure 49 Recursion



A_0 objects, both FPGAs. Ignoring the difference between mapping our A_2 object to the processor and the FPGA would easily lead to unbelievably sub-optimal results. For example if the A_2 object is an RDL unit which is truly nothing more than an AND-gate, it will be many orders of magnitude more efficient, if directly implemented in an FPGA.

12.2.6 Summary

In this section we defined the RDL mapping problem, including the constraints on a valid mapping and the goals for an optimal mapping. We also provided some background by discussing four standard IC CAD problems and their associated algorithms.

12.3 Algorithms

Given the similarity between RDL mapping and partitioning, it should come as no surprise that similar algorithms are applicable. In this section we discuss the criteria on which we can evaluate algorithms, or classes of algorithms, and the pros and cons of the algorithms themselves.

12.3.1 Criteria

There are several criteria which a good algorithm must meet.

First and foremost the algorithm must be capable of solving the problem outlined in Section 12.2. Of course some algorithms may solve useful subsets of the overall problem.

Second, since there is a human designer who can provide insight and guidance, the algorithm should be able to incorporate this information.

Third, human designers vastly prefer algorithms which produce predictable results. That is to say, changes in the input should produce roughly proportional changes in the output. Randomized algorithms, or those which rely on some form of ordering for their result, are thus shunned by both practical engineers and academics (mentioned in [71]).

Fourth, it is vitally important to be able to duplicate tool runs, for debugging both of the tools (algorithm) and the result. Randomized algorithms which use an LFSR-based random number generator can behave this way if the random seed is a user parameter. While commercial tools [15] do this, it is undesirable since the input to the CAD tool includes more than the actual design specification.

Finally in comparing algorithms from IC CAD problems there are three important differences. First, the RDL unit hierarchy has clear structure and often the mapping will be near trivial for a human engineer. Second, RDL primitives (channels and units) are significantly more heavy weight than IC primitives (transistors, LUTs and wires). Third, RDL provides a much more rigid, and thus analyzable, set of semantics than ICs, giving us better hope of accurate design metrics to capture the optimality of a mapping.

In the following sections we discuss classic CAD algorithms, general optimization algorithms and how they meet these criteria.

12.3.2 Fiduccia-Mattheyses

Because RDL mapping is so similar to IC partitioning, this section would be incomplete without mention of the Fiduccia-Mattheyses IC partitioning algorithm [39]. The simplicity of FM partitioning and the clarity of [39] have made this a baseline algorithm for most work since.

The FM algorithm is presented in terms of partitioning a graph into two clusters of limited size, while trying to ensure a minimum number of intercluster wires. The algorithm is based on the idea that switching some vertices from one cluster to the other will increase or decrease these metrics, and we wish to choose mostly "good" moves.

Unfortunately the extension of this algorithm to a heterogeneous set of clusters, which would be equivalent to different types and sizes of RDL platforms, is not clear. Furthermore, the algorithm does not seem to generalize well to include any notion of channel to link performance matching or the need to minimize the number of different designs by ensuring that different clusters (platforms) contain similar vertices (units). Finally, while simple constraints can be easily accommodated, more complex human sourced information such as channel traffic estimations and such, cannot be.

FM partitioning does not appear to offer a solution to RDL mapping. In fact FM fails on all but the reproducibility criteria given above, and is irrevocably optimized for IC scale problems not RDL scale problems.

12.3.3 Hierarchical

Because RDL is a hierarchical structural netlisting language, there is a chance to exploit the natural structure of a design specification when generating a mapping. It is a natural assumption that a series of RDL units which have been grouped to form a higher level unit belong on the same platform.

However, it is equally likely that two such hierarchical units which should be mapped orthogonal to the unit hierarchy. For example (drawing on RAMP), an array of processor cores and an array of L1 caches should not be mapped according to their hierarchy, rather each array should be split and CPU-L1 pairs should be mapped together. This kind of insight is the basis of hierarchical partitioning algorithms such as [20] and [65, 58, 40, 95].

None of these algorithms admit the possibility of using arbitrary information from a human designer. Nor do they easily extend to support any notion of channel to link performance matching. Most damning however, these algorithms fail to capture the need to minimize the number of different designs to minimize compilation time.

The need to minimize compilation time makes the extraction of regular structure a key goal, one which hierarchy analysis can help with. Ideally, most RDL specifications will be such that an $O(1)$ compilation time scaling is possible (*e.g.* 8 units per platform, or some simple divisor) but the hierarchical nature of this may not always be clear. Satisfying these constraints amounts to recovering, possibly convoluted, structure from RDL.

These heuristics all produce predictable results, and avoid random numbers, making this eminently suitable for CAD use. Furthermore, their primary purpose is to change the scale of the problem meaning that they should apply to RDL just as well as ICs.

A number of the above cited papers, [20] in particular, suggest heuristics which can be used to selectively expose the appropriate components of the hierarchy. We will mention the use of this in Section 12.4.1 to reduce the size of the RDL mapping problem, however for most of this section we will simply ignore the hierarchical nature of the RDL netlist. While this is likely to increase the problem sizes by destroying natural design groupings, it does not affect this work.

12.3.4 Simulated Annealing

Simulated annealing (SA) [59] is often the fallback algorithm for IC CAD tools, and many other complex optimization problems. This is primarily because it can be used on a wide range of $NP-hard$ problems, with a minimal knowledge of algorithms,

and little insight into the problem at hand. Simulated annealing allows a tradeoff between cost and benefit which can be controlled at tool runtime through the annealing schedule.

The downside of SA is that while it can handle the complexity of the goals in Sections 12.2.3, 12.2.4 and 12.2.5, is does so at the cost of a loss of information. Because the first step of SA is to deliberately throw away any structure in the input, SA necessarily starts at a disadvantage.

While a sufficiently slow annealing schedule will allow SA to recover a fully crystalline structure, and therefore exploit all possible regularity, there is no bound on the time it may take. SA is designed to allow the user to trade runtime against the optimality of the resulting solution. It is our expectation that the runtime of the RDL mapping algorithm will dwarfed by the runtime of the FPGA CAD tools, making this tradeoff somewhat useless.

SA is based on randomization, meaning it can produce unpredictable results and making it hard to duplicate tool runs for debugging.

Experience has shown that SA is best when a simple cost estimate metric can approximate the real cost.

SA is best when an approximate solution is acceptable, as this means the cost metric and annealing schedule need not be exact. The desire to share designs during RDL mapping, however implies that an approximate solution may be significantly worse than an optimal one. For example if each design takes 30 hours to compile, the difference between having a single design for all platforms and two designs is quite significant. Thus for RDL mapping, as for other problems, SA remains a viable way to obtain approximate solutions, but less than ideal.

12.3.5 Integer Programming

The combination of complex constraints and the $NP-hardness$ of the RDL mapping problem, suggest Integer Programming as a possible solution. Most constraints, can be expressed in terms of simple decision variables, while the optimization goals can be expressed in time or monetary terms.

The need for a unit to be mapped entirely to one platform means that integer programming or mixed integer programming will be required. The ease of composition of integer programming problems suggests that IP may be able to easily handle design recursion as mentioned in Section 12.2.5.

Most importantly, however, the form of IP constraints allows arbitrary human knowledge to be codified. Examples include the constraint that a certain unit must be on a certain platform, or a channel must be on a certain link. More complex examples include a constraint that two units must end up on the same platform, or that two platforms must share a single design.

Overall IP does very well on the criteria of Section 12.3.1. IP can solve the problem, incorporate human input and repeatedly produce predictable results. Furthermore, the relatively small number of RDL units (10,000) implies that it is not hopelessly large.

The major down side of linear and integer programming is the requirement that the constraints and goal be linear. This caused a fair amount of difficulty in our attempts to minimize the number of unique designs.

12.3.6 Summary

Table 6 Algorithms Report Card

Criteria	FM	Hier.	SA	IP
Applicability	F	C	B	A
Guidance	C	C	F	A
Predictability	?	B	D	B
Determinism	A	A	C	A
Scale	A	?	A	A

Despite the difficulties, successes in using IP for similar CAD problems such as [27] and especially [96] combined with the power of this algorithm suggest it as a good solution. The remainder of this section covers the complete formulation of RDL mapping as an integer program. Section 12.4 covers both the pre-processing and the actual IP formulation. Section 12.5 covers the implementation of an IP solver as a part of RDLC.

12.4 Solution

In this section we present the final algorithm we have designed to generate RDL mappings.

As with most $NP-hard$ CAD problems, we have chosen to decompose RDL mapping into the domain specific pre-processing followed by a generalized solver. We use a series of algorithms based on dynamic programming over compiler data structures to process the specific RDL description into an Integer Program. In concert with this work, we have developed an IP solver, described in Section 12.5 below.

The goal of our solution, presented in this section, is to take two hierarchical structural netlists, one of units and one of platforms, and produce a mapping from units to platforms. We wish to find

a minimum cost mapping where costs include space taken by the complete design, time to run the complete design and time to compile it using the post RDLC tools.

12.4.1 Preprocessing

Before any IP formulation there are necessarily a series of pre-processing steps to convert a general RDL specification into a useful form. Care must be taken to avoid a fully exponential problem growth during pre-processing. In particular there are many all-to-all pairings which require analysis before the IP can be written. These can be avoided through careful dynamic programming over the correct RDLC data structures.

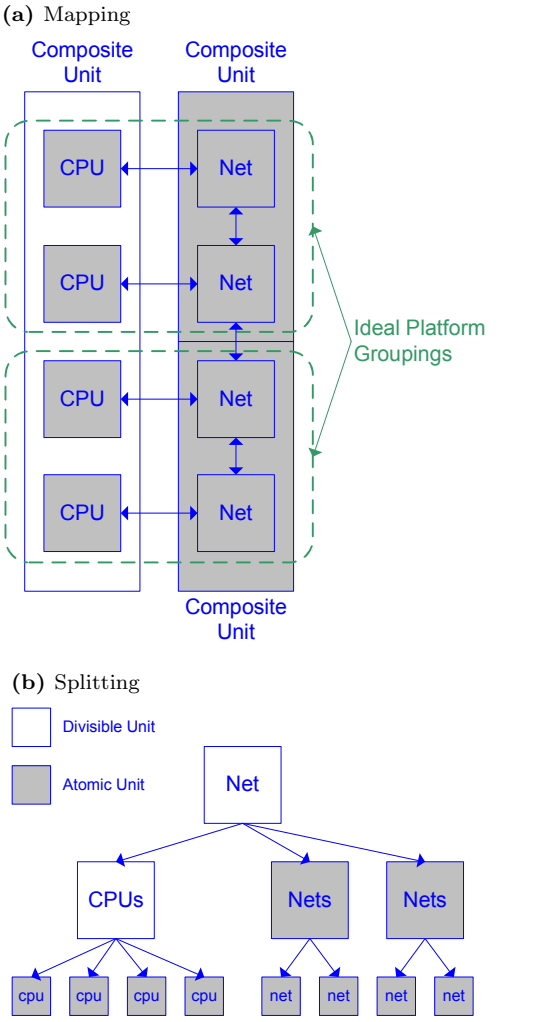
First, the RDL specification should be checked for viability at a coarse level. For example, units have resource (*e.g.* FPGA LUTs or PC memory) requirements, and it is relatively easy to check that there are enough total resources for all units. RDL syntax and semantic checks fall into this category. These tests should be designed to ensure that we do not attempt to solve RDL mapping for infeasible designs.

Second, as suggested in our discussion of hierarchical partitioning (Section 12.3.3) it likely that we will selectively exploit hierarchy. The basic idea is that some hierarchical units (platforms) will be split, exposing the units (platforms) of which they are composed, and some will be left atomic as shown in Figure 50. This may allow us to significantly reduce the problem size, by exploiting the intuition of the human designer. We have left this step and its design to later work.

Third, the RDL units and platforms must be grouped into equivalence collections. By using a pair of Union-Find data structures or perhaps simple hash tables for dynamic programming this can be performed in polynomial time. Because RDL includes platform and unit parameterization, determining equivalence is an interesting semantic problem. For example RDL platform in the first two compiler and language revisions (RDLC1 and RDLC2) do not include I/O specifications. In order to determine platform equivalence, this information is necessary and so is being added to the third language and compiler revision: RDLC3 (see Section 16.4.1).

In this section we have outlined the preprocessing steps, beginning with viability checking, which acts as a filter to avoid unnecessary work, and culminating with the creation of platform and unit equivalence collections. The end result of preprocessing consists of the unit and platform equivalence collections, along with size and performance metrics for each element of these collections, and a variety of user constraints.

Figure 50 Hierarchy Splitting



12.4.2 Integer Program

In the remainder of this section we give the rules by which an instance of the RDL mapping problem is converted to an integer program.

We will not restrict ourselves to one of the canonical IP forms, as well known manipulations can be used to convert between them. For example, we will make use of equality constraints, as well as double ended inequalities. We will minimize an objective function.

In the remainder of this section we will denote the set of units U, platforms P, designs D, channels C and links L. We will denote the set of unit equivalence collections UEC and platform equivalence collections PEC. Note that to allow for the

situation where each platform requires a different design, for a given host (set of platforms) we introduce designs such that $|P| = |D|$.

12.4.3 Encodings

The vast majority of LP and IP constraint encodings, are commonly known. For example an equality constraint becomes two inequalities. A difference constraint becomes two inequalities (which together form an equality constraint) with a slack variable to pick up the difference.

In this section we introduce several encodings for decision variables, which are vital to our work. A decision variable in an integer program is a variable x_0 constrained to the set $\{0, 1\}$.

We will need to perform simple Boolean operations such as "and" ($\wedge$), "or" ($\vee$) and "not" ($\neg$) over decision variables. What makes this difficult is that we must avoid the use of the min (or max) objective to obtain these binary operations, as any change in the objective will skew the resulting program.

Encoding "not" ($\neg$) is simple. Given two decision variables $x_0, x_1 = \{0, 1\}$ we can use the constraint $x_0 + x_1 = 1$ to encode $x_0 = \neg x_1$.

Encoding "and" ($\wedge$) as well as "xor" ($\oplus$) is almost as simple. Given four decision variables $x_0, x_1, x_2, x_3 = \{0, 1\}$, we can encode $x_2 = x_0 \wedge x_1$ and $x_3 = x_0 \oplus x_1$ using the constraint $x_0 + x_1 - 2x_2 - x_3 = 0$.

With an encoding for both "not" ($\neg$) and "and" ($\wedge$) we can encode "nand" which is a universal Boolean function. This means that we can encode any Boolean function of arbitrary complexity with a linear expansion in the number of variables and constraints. However, in the following formulation we will wish to compute "or" ($\vee$) over very large collections of decision variables. Thus we provide an efficient (sub-linear) encoding.

Given five decision variables $x_0, \cdots, x_4 = \{0, 1\}$, we can use two constraints to compute the $x_0 \vee x_1$. The constraint $x_0 + x_1 - 2x_2 - x_3 = 0$ will enforce $x_2 = x_0 \wedge x_1$ and $x_3 = x_0 \oplus x_1$. Notice, however that $x_2 \wedge x_3 = false$ and that $(x_2 \oplus x_3) = (x_2 \vee x_3) = (x_0 \vee x_1)$. Thus we can add another constraint $x_2 + x_3 - x_4 = 0$ to encode $x_4 = (x_2 \oplus x_3) = (x_2 \vee x_3) = (x_0 \vee x_1)$.

This encoding can be easily generalized to arbitrary length "or" clauses. The first constraint above ($x_0 + x_1 - 2x_2 - x_3 = 0$) can be generalized to $x_0 + x_1 + \cdots + x_n - 2^m y_m - 2^{m-1} y_{m-1} - \cdots - y_0 = 0$, where $m = \lceil log_2 n \rceil$. This general constraint encodes $|\{i | x_i = true\}|$ as the bitwise concatenation $\{y_m, y_{m-1}, \cdots, y_0\}$. The trick is that the $(x_0 \vee x_1 \vee \cdots \vee x_n) = (y_0 \vee y_1 \vee \cdots \vee y_n)$. Thus be repeated use of such constraints we can eventually reduce the number of variables we need to "or" from n to 2, at which point the second constraint from above (the "and-xor" constraint) can be used.

The space bounds on these encodings are vitally important to keeping the encoding of RDL mapping as an IP a polynomial time reduction. Furthermore, as we hope to produce a useful CAD tool, efficient encodings are important to minimize run times.

12.4.4 Correctness

The correctness constraints for RDL mapping are simple.

Each unit must be mapped to exactly one platform. For this we introduce a series of decision variables $\{up_{ij} | (0 \leq i < |U|) \wedge (0 \leq j < |P|)\}$ which indicate the unit i has been mapped to platform j. We add constraints $\forall i (\sum_{0 \leq j < |P|} up_{ij} = 1)$.

Each channel must be mapped to exactly one link. We introduce decision variables $\{cl_{ij} | (0 \leq i < |C|) \wedge (0 \leq j < |L|)\}$ which indicate the channel i has been mapped to link j. We add constraints $\forall i (\sum_{0 \leq j < |L|} cl_{ij} = 1)$.

A channel which connects two units must be mapped to a link which connects the platforms to which the units have been mapped. For each channel i connected to units a and b, for each link j connected to platforms c and d we add three constraints $2cl_{ij} - up_{ac} - up_{bd} - 2x_0 = 0$, $2cl_{ij} - up_{ad} - up_{bc} - 2x_1 = 0$ and $x_0 + x_1 = 1$ where $x0, x1 = \{0, 1\}$.

We may also have constraints that units are allowed or disallowed on certain platforms, and similar restrictions for channels and links. These constraints can be expressed by forcing the value of certain decision variables. For example, forcing unit i to be mapped to platform j is merely a matter of adding a constraint $up_{ij} = 1$. In general such constraints may then be simplified out of the IP before solving.

Depending on the actual platforms involved there may be several other constraints, primarily dealing with resources. For example, an FPGA platform has a limited number of LUTs. We can express these constraints quite easily as $\forall j (\sum_{0 \leq i < |U|} c_{ij} up_{ij} <= max_j)$, where max_j is the resource limit of the platform and c_{ij} is the cost of unit i on platform j. Notice that we can easily have such constraints for multiple resources, *e.g.* FPGA LUTs, RAM, External Interfaces. Also units can have different resource costs on different platforms, reflecting the RDL feature which allows unit implementations to be platform dependent.

12.4.5 Designs

Each platform must instantiate exactly one design. We introduce decision variables $\{dp_{ij}|(0 \leq i < |D|) \wedge (0 \leq j < |P|)\}$ which indicate the design i is being instantiated on platform j. We add constraints $\forall j(\sum_{0 \leq i < |D|} dp_{ij} = 1)$.

In order to minimize the number of designs which must be compiled, we must find some way to express the number of designs. We introduce a set of decision variables $\{used_k|0 \leq k < |D|\}$ which indicate, for each design, whether it is used at all or not. We add a constraint for each design $\forall k \ used_k - \vee_{0 \leq j < |P|} dp_{kj} = 0$. We can then introduce a variable to count the number of designs $used' = \sum_{0 \leq k < |D|} used_k$.

Minimizing the number of designs is then merely a matter of minimizing $used'$. This can be traded off against other optimization objects by the use of a cost coefficient in the IP objective function.

In order to account for varying compilation costs (*e.g.* that FPGAs are hard to compile for, but PCs are easy) of different PECs we need a more complex formulation. First, we may divide the design set D into subsets or DECs, one per-PEC. We then modify the correctness constraint of the previous section to ensure that only designs of the correct DEC are instantiated on platforms from the correct PEC. We can then add a set of decision variables $\{used_l|0 \leq l < |PEC|\}$ which indicate, for each DEC how many designs in that DEC are used along with the constraint $used_l = \sum_{\{k|(0 \leq k < |DEC_l|) \wedge (D_k \in DEC_l)\}} used_k$.

Similar to the resource usage constraints of platforms, we may also wish to use per-unit resource costs to compute variables which indicate the resource usage of a design. While this is not necessary for correctness, since the platforms are already resource constrained, it would allow the optimization objective to include resource usage dependencies. For example compilation time for FPGA designs might be proportional to the number of LUTs used in the design, or with some different constants it might be proportional to the fraction of the FPGA used. We will leave these generalizations for later work, when we can test with real world examples.

12.4.6 Optimality

For the purposes of this section we will explain the formulation of several optimality objectives. The fact is that until we have a more complete RDLC infrastructure, as well as example designs it is unclear exactly what these objectives should be.

In general the optimality of a mapping, as described in Section 12.2.3, depends both on the number of resources (platforms and links) to implement the mapping and the time it will take the resulting system to simulate a unit of real time. Recall that RDL is a language primarily for describing gateware-based hardware simulators, where the resource and virtualization overhead may be traded. In order to trade the two certain information will be required of the user, including the length of real time which the simulator will be needed for and the cost of using additional platforms either in time or dollars. Absent an in-depth investigation of the needs of RAMP researchers and RDL users, we will avoid a discussion of the constants in this section.

Resource usage, in the form of platforms used is easy to measure, in a similar manner to design count minimization in the previous section.

Minimizing simulation overhead is a matter of minimizing the largest discrepancy between a channel timing model and the underlying link to which the channel is mapped, as shown in Figure 51. Note that, in this case we refer only to the cost when the channel is higher performance than the link, as the reverse case costs resources, but not time. There are two ways to incur timing overhead, either when all of the channels sharing a link have a higher aggregate bandwidth than the link, or when a channel has lower latency than the link it is mapped to.

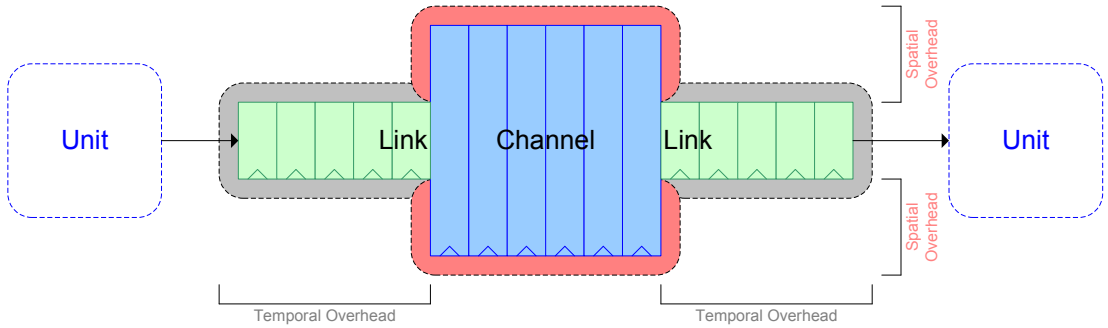
It is a simple matter, though beyond this section , to compute the bandwidth requirement for any channel i, which we will call cbw_i. It is equally easy, though link dependent, to compute the bandwidth allowed on any link j, which we will call lbw_j. We then introduce two variables $0 \leq bw_j^+$ and $0 \leq bw_j^-$ representing the unused and over used bandwidth of link j respectively. A constraint of the form $\sum_{0 \leq i < |C|}(cbw_i cl_{ij}) - lbw_j - bw^- + bw^+ = 0$ for each link j will complete the formulation.

Given the bw_i^- variables, we wish to minimize some linear function of the largest of these, rather than the sum since over used bandwidth translate directly into simulation latency, which will overlap. We can then add a variable bw^- and constraints $\forall j \ bw^- \leq bw_j^-$. By minimizing bw^-, we can minimize the simulation latency which results from overused link bandwidth.

We can also compute per-channel and per-link latencies, $clat_i$ and $llat_j$. We then introduce two variables $0 \leq lat_{ij}^+$ and $0 \leq lat_{ij}^-$ representing the excess and insufficient link-channel latency respectively. We add constraints $\forall i, j \ (clat_i < llat_j)$ of the form $(llat_i - clat_j) \leq lat$. The variable lat is then the maximum additional latency, and by minimizing it we can minimize the simulation latency which results from mapping low latency channels to high latency links.

The variables bw^-, lat and the design usage variables of the last section together allow us to

Figure 51 Channel/Link Optimality



trade all forms or resource, runtime and compilation time. Knowing the amount of simulation time the designer intends to use will allow us to make this tradeoff, but it remains unclear how RDL users and RAMP researchers will want to specify this information.

12.4.7 Recursion

Recursive levels of design (Section 12.2.5) imply that the platforms (links) of a higher level design are the units (channels) of a lower level design. By introducing parallel sets of variables and constraints as outlined above, we can easily extend our IP formulation to include such designs.

While arbitrary human constraints may help single level designs, they are incredibly powerful when combined with design recursion. For example, the designer may be able to specify constraints which span different levels of abstraction.

By optimizing the complete design and allowing cross-abstraction constraints IP provides a powerful general formulation which none of the other algorithms in Section 12.3 can aspire to.

12.4.8 Solution

In this section we have given the formulation of RDL mapping as an integer program. We have given various basic encodings, correctness constraints and optimization goals, including formulations of all aspects of the RDL mapping problem. The problem reduction specified in this section is polynomial, and quite simple, meaning it can, and will, be automated as part of RDLC.

12.5 Implementation

In the previous sections we have discussed the RDL mapping problem, several candidate algorithms and finally our formulation of the problem as an integer program. In this section we describe an IP solver built on the same Java framework which is the basis of RDLC3 (see Section 16.4.1). While the integration of RDLC3 and the IP solver is not complete, this has been an important step towards a complete RDL mapping tool.

12.5.1 Branch and Bound

There are many algorithms for solving integer programs; we chose branch-and-bound as being one of the simpler ones. Because integer programming is $NP - complete$, it is widely accepted that there is little hope of a polynomial time solution, thus the best we can hope for is a decent heuristic.

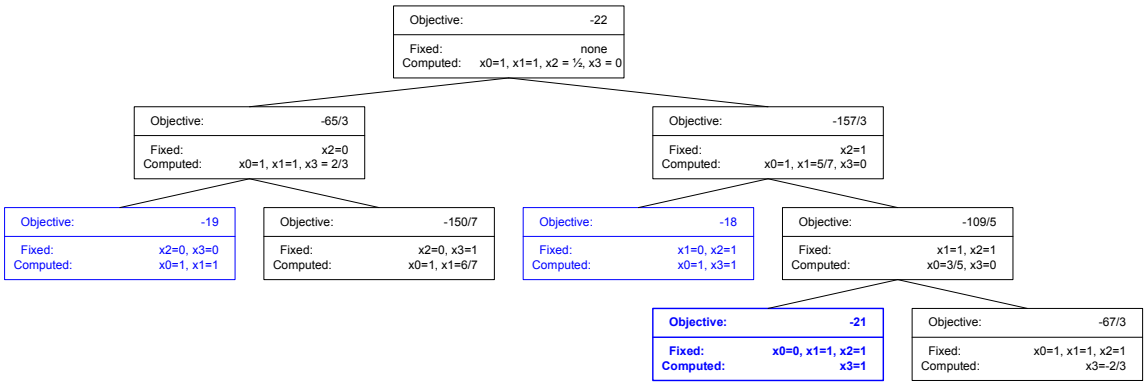
Branch and bound for IP solving involves repeatedly solving a tree of linear relaxations of the original integer program, as shown in Figure 52. In the first step, we simply solve the linear relaxation. Should the result be an integer solution, we are, of course, done.

However in the event that result is a fractional solution, we branch to two sub-problems, fixing one of the fractional variables in the process. We force the variable to the ceiling of its fractional value in one sub-problem, and the floor in the other. Both of these problems then have one fewer variables, and we recurse, solving each as a linear relaxation.

In order to reduce the running time, we do not recurse on sub-problems which are rendered infeasible by the forcing of a variable. Because the problem is highly constrained, we expect this to noticeably reduce the running time as many solutions will be infeasible. Nor do we recurse on sub-problems which cannot improve on the current best integer solution.

Our branch and bound solver uses a skiplist to keep track of the active sub-problem with the best current optimization result. We also keep track of the parent sub-problem so that at the end we can reconstruct the values of all the variables, which are needed to generate the RDL mapping. Our imple-

Figure 52 Branch & Bound



mentation is less than 500 lines of Java, including test cases thanks to RCF, a library we have been developing for RDLC3 (see Section 16.4.1).

Branch and bound also has the bonus that the individual linear relaxations can provide quality bounds on the ultimate solution. These bounds can be presented to a human designer who may wish to stop the algorithm should there be little hope of a solution with the quality they desire.

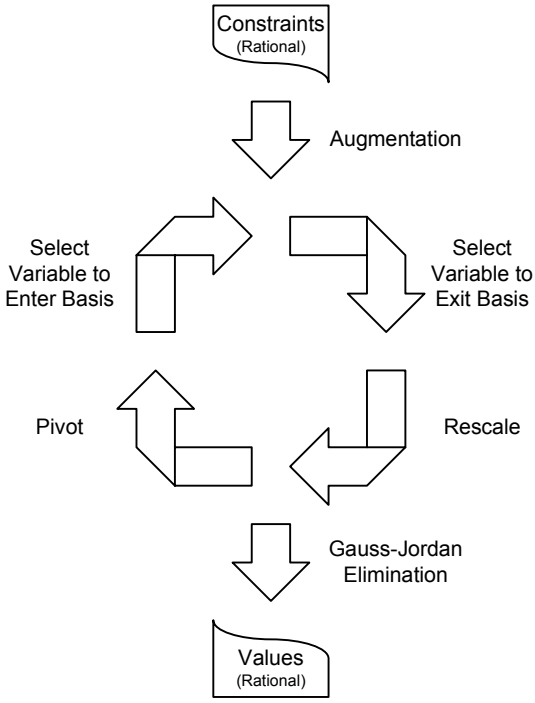
12.5.2 Simplex Solver

Branch and bound is based on the repeated solving of linear relaxations, meaning that it relies on an LP solver. Rather than attempt to code and test a complex LP solving algorithm based on interior points or the like, we chose the simplex algorithm as shown in Figure 53. The main point of this work is to produce a usable tool, and given the 2 to 30 hour FPGA compile times, spending a few extra minutes solving linear programs is insignificant. Not to mention the fact that a working, though slow solution will be usable, both for RDL users and as a reference implementation, during the period in which we investigate better algorithms and problem specific heuristics.

Our simplex solver is designed to solve problems of the form $min(cx)$ subject to $Ax \leq b$ and $x \geq 0$. The simplex algorithm itself is well known, and we will avoid describing it here.

Our code amounts to another 500 lines of Java code, which includes a Gauss-Jordan elimination routine to recover the actual variable values after solving, and a few test cases. Our implementation is based on a separate rational numbers package which we have developed to be a general part of RCF. In truth, while our tests are all based on rational numbers with 32bit integers, the algorithm is generalized and can work with arbitrarily large numbers.

Figure 53 Simplex



12.6 Conclusion

As of now, a human designer is required to specify which platforms implement which units, in other words the mapping from units to platforms. While RDL makes the specification of this mapping concise and simple, designing it remains difficult even for highly regular structures.

We have analyzed the problem of RDL mapping in the context of similar $NP-hard$ problems from IC CAD in order to better define it. We have presented multivariate optimization algorithms, and evaluated them as they apply to the RDL mapping problem. Finally we have given a complete formu-

lation of RDL mapping as an integer program and briefly described our implementation of a brand-and-bound/simplex IP solver.

In a comparatively short period we have sketched the problem and solution, and are currently in the middle of building a complete tool for generating RDL mappings. The vast majority of the work was in defining the problem and finding a satisfactory formulation. Thanks to our code framework, the actual IP solver implementation turned out to be quite reasonable. For more information about current and future work, please see the next section.

12.7 Future Work

While we have solved the most pressing problems in the path of building an automated RDL mapping tool, some remain.

Completing the test cases for the simplex, Gauss-Jordan elimination, and brand-and-bound algorithms remains extant.

While we presented a very complete IP formulation in Section 12.4.2, the optimality metrics leave a bit to be desired. In particular, it has been suggested by RAMP researchers that analysis of real-world traffic patterns may be highly beneficial. For example, some channels may be rarely used, whereas others may remain at bandwidth capacity. This information, and even temporal correlation of traffic patterns may change the optimal RDL mapping significantly. In the future we hope to find a way of incorporating this information into our formulation.

We would also like to find a way to analyze platform dependent unit equivalences. Two units may be equivalent on one platform (for example Java which is easily parameterized at run time), but vastly different on another (for example an FPGA which cannot be parameterized after compilation). This should be a relatively simple extension to our formulation and pre-processing.

Finally, we would like to add support for complex network types to our formulation. RDL channels are strictly point-to-point, but links are not, meaning that busses and multicast networks at the target (unit) level may be completely subsumed by a single link. In the context of recursive designs this may have serious implications should our formulation completely ignore this fact.

The main thrust of the future work on this project will be to apply the algorithm described herein to real RDL designs to form a practical tool. Aside from collecting complex RDL designs which exercise the various parts of the algorithm and will generate useful performance numbers, the majority of this work will be implementation. Implementation tasks range from abstracting Gauss-Jordan elimination as a general algorithm, to writing code to translate real RDL designs into integer programs. The fact is that CAD tool integration is difficult, and that the constant factors in the running time matter quite a bit for practical applications.

The biggest problem in writing this extremely useful tool, is the relatively brittle code base of RDLC2. In particular the problems with parameterization, and the complexity of the internal AST data structures seriously complicated our attempts. We believe that a compiler rewrite (see Section 16.4.1) is likely necessary before this tool can be completed and useful.

Chapter 13

Fleet

In this section we present the design and operation of a novel computer architecture we have been developing, called Fleet, along with the development of a simulator for it, developed in the RAMP Description Language (RDL). This is intended as a walk-through for a designer wishing to experiment with Fleet, and includes an architectural description (see Section 13.1), a guide to the code (see Section 13) and instructions for building and programming a Fleet. Not only is Fleet a novel architecture, but this work represents the first hardware implementation of a complete Fleet processor capable of executing code, though there have been circuit-test ASICs with the name Fleet.

It is interesting to note that while this project has successfully made good use of RDL it is not part of the main RAMP effort. Though it is obviously a hardware projects with some element of computer architecture, there is quite a bit of variety when compared to the manycore experiments which are the bastion of RAMP. This shows some of the main strength of RDL, though clearly rooted in the needs of RAMP, it has proven useful on a slightly wider range of applications.

13.1 A New Architecture

Fleet [31, 85, 86, 87] is a novel computer architecture based on the idea that the ISA should be focused on exposing the hardware's abilities to the programmer, instead of hiding them below abstractions. Most notably, in the years since the emergence of class CISC and RISC architectures, there has been a cost inversion in IC design between wires and transistors. Transistors, once the most expensive part of a CMOS IC, are now often considered free as they fit easily below the massive amounts of on chip wires necessary to implement busses and other higher performance interconnects. Taking this a step further, we have worked for 30 years to guarantee sequential operation of seas of transistors, only to realize now that we want concurrent processors in the end and that sequentiality is costly and counter productive.

13.1.1 ISA

A classic ISA is focused on the storage (register file or memory) and operation (ALU, *etc.*) operations relying on the hardware designer to be clever about making these things happen. This was a reasonable approach as it allows an assembly language programmer to easily translate an algorithm, and a hardware designer to optimize these expensive transistor heavy operations.

Fleet is based around collections of concurrent instances of a single instruction: move. The ISA thus focuses on the movement of data, the expensive operation given the relative cost of transistors and wires, allowing the high performance hardware designer and programmer to cooperate rather than working against one another. Of course storage, computation and sequentiality are still necessary, but these operations in Fleet are encoded in the locations to which data items are moved and dependencies between moves.

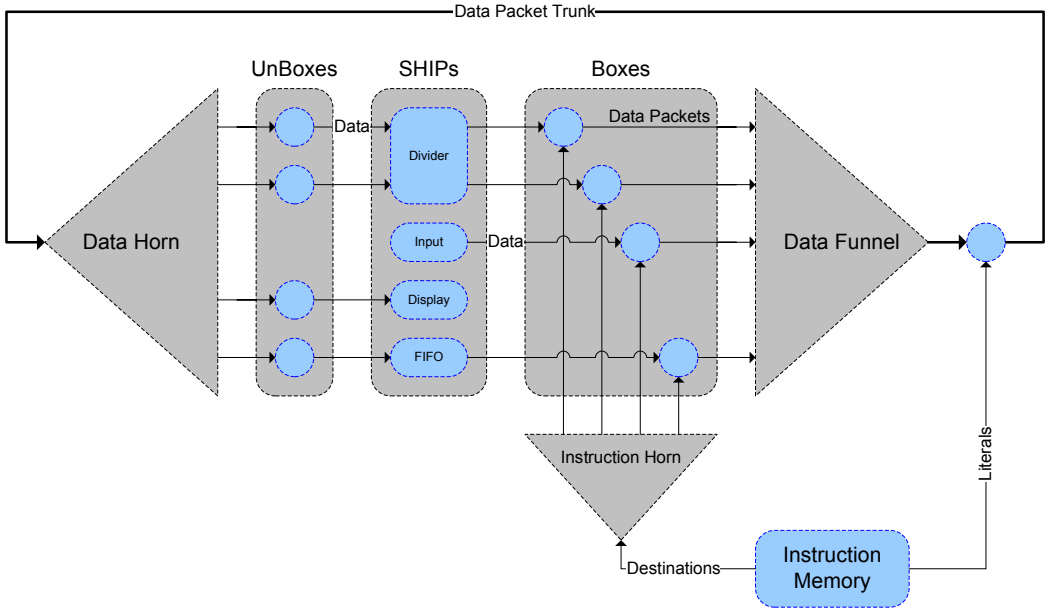
In many ways Fleet turns the job of micro-architectural scheduling over to the compiler or programmer rather than forcing a architect or hardware designer to make the relevant decisions. In essence one might view a Fleet processor as a standard Tomasulo or Out-of-Order processor core, from which the forwarding logic, score-boarding and such logic have been removed. This shift means that a Fleet processor can easily be more efficient, particularly for streaming applications, at the cost of a more advanced compiler or a better assembly language programmer.

Figure 54 shows a high level hypothetical block diagram of a Fleet processor. A Fleet processor consits of a collection of operators over data words called Ships[1], and some form of switch fab-

[0]The ideas behind Fleet are in main the work of Ivan Sutherland, and numerous others including Adam Megacz and Igor Benko.

[1]If you don't like nautical themed jokes, we recommend

Figure 54 Top Level Fleet



ric or NoC which connects them to deliver data. While Figure 54 shows simple "horn-and-funnel" switch fabrics for the data and instructions, this is for explanation and testing purposes only and any real implementation would likely have switch-fabric without such obvious bottlenecks.

Ships are connected to the switch fabric by intermediaries labeled boxes and unboxes[2] in Figure 54 which are responsible for adding and removing routing headers respectively. A Fleet is programmed by delivering move instructions, that it routing headers, to the "boxes" which will then concatenate them with the relevant data and send it on. Thus a one-shot move instruction `move [1] Adder.Sum -> Display.Input;` would be translated to a route from `Adder.Sum` to `Display.Input`, and then delivered to the box for Adder.Sum to be joined with the proper data. We thus say that there is a second switch-fabric dedicated to moving instructions from the "Fetch Ship" to the relevant boxes.

Because instructions in Fleet are no longer sequential operations to be performed on a static register file, Fleet is able to expose the available concurrency of the underlying hardware quite easily. Collections of move instructions are called "bags" for the simple reason that the instructions within a bag may be executed in any order, and multiple bags may be active at any one time. This gives the fetch hardware designer and the application coder, or more likely the compiler, a significant amount of flexibility. For example the designer of the fetch Ship may decide to deliver instructions sequentially, in the order they are found in the cache or all at once if the instruction switch fabric has enough bandwidth.

Sequential operations are encoded by the use of tokens, pieces of data with no value or at least no useful value whose arrival denotes an important event. Thus the result of one operation can be used to trigger the loading of an additional codebag, thereby ensuring that none of the instructions in the new codebag can be executed before that token arrived. In order to make this feasible move instructions may have more than one destination, allowing the result of an operation to be send to multiple Ship inputs.

It is worth mentioning that nowhere in the above section was a list of Ships given. Fleet is an architecture, not a particular processor, and different implementors with different applications are free to select the Ships they need, as well as the exact topology of the switch fabric. Fleet is characterized by its expression of concurrency through unordered source code, and its focus on efficiency through controlled movement of data.

13.1.2 Assembly Language

Given that the ISA of Fleet only has one instruction, the assembly language would seem trivial. However, the `move` instruction has several embellishments, all of which are shown in Program 61. In

skipping this section.

[2] We use this term for consistency with our code, though the proper term is now "Dock."

the remainder of this section we will walk through this simple example line by line.

Program 61 A Simple Fleet Program

```
1 initial codebag Bag1 {
2   move (1) -> Display.Input;
3   move (Bag2) -> Fetch.CodeBag;
4   move (true) -> FIFO[0].Input;
5   move FIFO[0].Output -> Display.Input;
6 };
7
8 codebag Bag2 {
9   move (2) -> Display.Input;
10  move (token) -> Display.Input;
11  move [] IntegerInput.Output ->
        Display.Input;
12 };
13
14 initial codebag Bag3 {
15  move (3) -> Display.Input;
16 };
```

First of all, Program 61 declares three codebags of move instructions starting on lines 1, 8 and 14. Each one of these code bags contains a logically unordered group of move instructions which may be executed concurrently, that is with any convenient ordering or parallelism. The code bags `Bag1` and `Bag3` are also marked `initial` meaning that they should be fetched and executed immediately upon processor reset.

Looking at `Bag3` first, it contains a single move instruction `move (3)-> Display.Input;`. The keyword `move` marks this as a move, the `(3)` denotes the literal value 3, and `Display.Input` denotes the port `Input` on the `Display` Ship. For those accustomed to RDL, the syntax of Ship port names matches exactly what would be written in the RDL to name those ports.

`Bag1` on line 1 is more complicated including three literal moves, an integer (line 2), a codebag address (line 3) and a boolean (line 4). Finally line 5 shows a standard move from the `Output` port of the `FIFO[0]` Ship, which is the first Ship in the `FIFO` array using RDL unit array syntax.

`Bag2` is interesting for line 11 which shows a so-called standing move, indicating that all data from the `Output` port on the `IntegerInput` Ship should be sent to `Display.Input`. There is also an example of a token literal on line 10, which is a piece of data with no useful value. Aside from this it is worth noting that all three of the move instructions in `Bag2` will send data to the display, and because the moves are concurrent there is no guarantee what order the data will appear in.

13.2 Documentation

This section documents both the code and the tools necessary to instantiate a Fleet processor with some desired set of Ships and compile a small assembly program to run on it.

In order to allow experiments with a variety of switch fabrics, Ships and Ship configurations, we have created a gateware model of Fleet using RDL. We have implemented all of the relevant components shown in Figure 54 in Verilog, and created RDL unit descriptions for them. Creating a simulation or emulation of a Fleet therefore is a matter of modifying a bit of RDL to specify the list of Ships, and then mapping (see Section 8.4) the design to an FPGA or HDL simulator.

In the remainder of this section we will document all of the RDL units which are combined to form a Fleet. We will also document the integrated compilation (see Section 13.2.4) process with will compile the Fleet and assemble a program for it simultaneously. Finally we will show several simple example Fleet programs written in the assembly language we have developed (see Section 13.1.2).

13.2.1 Unit: `Fleet`

The `Fleet` unit represents the top level of the design and instantiates the `Ships` unit along with all of the, automatically generated, horns and funnels for a simple switch fabric. In addition it provides the basic channel connections required in all Fleets. This unit has several parameters listed below which define things like the machine word width, and are necessary for the Fleet assembler.

AWidth: The bitwidth of Ship port addresses. This determines the number of Ships which can be added to this Fleet, as well as the size of the binary instruction representations.

CWidth: The bitwidth of the count on counter moves. In addition to one-shot and standing moves, move instructions can have a counted bound, in which case they will move as many pieces of data as specified. This value determines the maximum number of data items a move instruction can move.

DestCount: The number of different destinations any one move instruction can have. This determines not only which instructions can be validly encoded, but also the bitwidth of their encoding.

IWidth: The bitwidth of integers in this Fleet; *i.e.* the machine word width.

MemAWidth: Bitwidth of the instruction memory addresses. This determines the maximum number of move instructions which can be held in memory at one time.

CBOffWidth: The length of the bag offset field in a code bag descriptor. Each code bag descriptor is `MemAWidth` bits long, some of which specify the base address of the code bag, and some specify the length.

Because it is the root of Fleet descriptions the `Fleet` unit has to input or output ports[3].

13.2.2 Unit: `Ships`

The `Ships` unit contains all the Ship instances, drawn from the below pool. By using an RDLC2 plugin (see Section 6.5) the Ships instantiated here are automatically connected to the switch fabric. Thus to add a new Ship, in other words a new functional unit, to a Fleet, all that is required is to instantiate it in the `Ships` unit and set any relevant parameters. Many copies of the same Ship may be instantiated as an RDL unit array (see Section 6.2.3) as shown with the `FIFO` unit array on line 5 of Program 61.

All Ship units must have only ports which are compatible with the Fleet in RDL datatypes. In particular there are types for: Integers, Booleans, Tokens and CodeBag Descriptors. There is also a type for variant Ships which can accept data of any type, such as the FIFO (see Section 13.3.4) and Rendezvous (see Section 13.3.5).

In Section 13.3, we describe the details of ports and operation of the various Ships which we have implemented.

13.2.3 Switch Fabric

The switch fabric shown in Figure 54 consists of boxes & unboxes and horns & funnels. Boxes and unboxes are responsible for converting data between switch fabric messages, which include routing headers and simple data to be presented to the Ships. The horns and funnels provide a basic switch fabric implemented as a binary tree for simplicity, definitely not for performance. These are in turn constructed from simple binary fork and join operations.

Boxes are the units responsible for concatenating a piece of Fleet data with a destination address, which will then be used by the data horn. Boxes are also the destinations of move instructions, as specified by the source address listed in the instructions. In addition boxes provide support for multiple message destinations by reusing the data from the Ship to form multiple switch fabric messages. In order to ensure a consistent encoding for instructions, each instruction has multiple destinations, some of which or all of which may be the bit bucket. This is a special destination, which does not physically exist, leaving the boxes to simply drop any data bound for it. Thus the same mechanism whereby the programmer discards data is used by the assembler, all without incurring and power or time to move useless data about.

The unboxes are, by comparison, quite simple and responsible only for stripping the address from each message from the switch fabric and delivering the remaining data to a Ship.

The switch fabric is implemented by a pair of generic RDL units: `Horn` and `Funnel` neither of which contains any unit instances. Instead their implementation is filled in by a pair of plugins which generate binary trees of the appropriate width using the `Fork` and `Join` respectively. Note that these are sparse trees, meaning that with only 7 destinations anything addressed to destination #7 will be routed to the highest actual destination, #6.

The `Join` unit is a simple two way priority arbiter[4] for Fleet switch fabric messages. `Fork` simply checks a predetermined bit of the address on each message and routes the messages to the correct output based on that value.

While it has been suggested multiple times in this document, and it is obviously necessary, that the switch fabric be implemented more efficiently for a real Fleet processor, these simple implementations have some virtues. Obviously literal constant injection is somewhat simplified by being able to tap in to the switch fabric trunk as shown at right in Figure 54. More importantly, however, these implementations can't reorder data, a subtle but important restriction on the initial Fleet designs. This may be reconsidered in the future, since it heavily restricts the design of the instruction switch fabric.

13.2.4 Launching

In order to actually experiment with a Fleet, one must compile an RDL description of the desired Fleet and assemble some code for it. We taken advantage of the plugin architecture of RDLC2 (see Section 10) to integrate this process with RDLC itself (see Section 9.3), making the process quite simple. Shown in Figure 55 is the internal toolflow from Fleet assembly program to complete source code output, including an instruction memory initialized to contain the program. Highlighted in red

[3] A researcher interested in multi-core Fleets, or Flotillas, might change this one day.

[4] Deadlock is the programmers responsibility in Fleet.

are the parts which are unique to Fleet, and are added to the normal RDLC2 mapping flow (see Section 31).

In particular the Fleet compiler is actually a highly parameterized Fleet assembler, because it takes not only a Fleet assembly language program but an RDL description of the hardware to run it on. The parameters listed in Section 13.2.1 determine not only the structure of the processor, but of course the binary encoding of the instructions which it must generate.

Aside from the main command plugins there are several other specialized Fleet builder RDLC2 plugins. In particular the horns and funnels are generated by a pair of aptly named plugins which are responsible for creating the binary tree structure. This could of course easily be replaced by generators for a different switch fabric topology.

Finally the Ships in the `Ships` unit are connected to the boxes by a plugin designed for this purpose. The connector plugin will search all of the RDL unit instances inside of the `Ships` unit for RDL ports which are of the proper type, and unconnected. This hides all the details of connecting ports properly based on their type, and managing the assignment of port address to Ship ports. Thus a Fleet program which is compiled directly to a Fleet never needs to contain numeric aliases, as they can be extracted from the RDL source code.

The correspondence between Ship ports and switch fabric addresses, as well as the result of the Fleet compiler stage of the toolflow combine to produce a memory image containing the assembled code. This memory image is then loaded in to the read-only instruction memory through the usual RDLC2 memory generation plugins (see Section 10.4.3).

In the end the code produced from this process is a Verilog description of a complete Fleet with an initialized instruction ROM. This description can be tailored, in RDL, for either an FPGA or HDL simulator implementation.

13.3 Ships

In this section we describe the details of ports and operation of the various Ships which we have implemented. This is not intended as a complete list of Ships, as it is not clear what applications they can or cannot efficiently support. Instead this is merely a small example of the kinds of Ships we envision, and those necessary to run the examples shown in Section 13.4.

13.3.1 Unit: `Adder`

The adder Ship, whose ports are listed in Table 7, is fairly self explanatory. What is notable are the firing rules, which state that it will perform one addition for each pair of inputs and consume them in the process.

Table 7 Adder Ports

Dir	Type	Name	Description
In	Integer	Adder	One of the two values to be added
In	Integer	Addend	One of the two values to be added
Out	Integer	Sum	The sum of the two inputs

13.3.2 Unit: `Comparator`

The comparator Ship, whose ports are listed in Table 8, takes two integers and produces a boolean. The input ports are labeled according to which one should be larger for the output to be `true`.

Table 8 Comparator Ports

Dir	Type	Name	Description
In	Integer	Small	The smaller of the two inputs
In	Integer	Large	The larger of the two inputs
Out	Boolean	Result	True if the larger input actually is

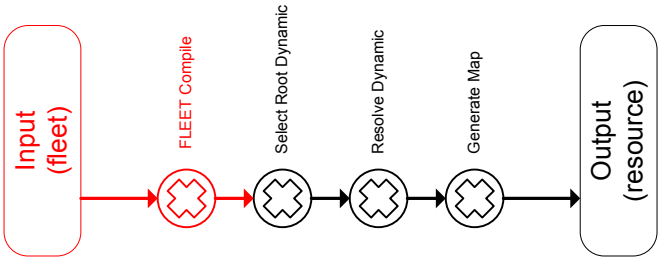
13.3.3 Unit: `Multiplexor`

The multiplexor Ship, whose ports are listed in Table 9, takes two variants and a boolean and sends one of the variants to its output.

Table 9 Multiplexor Ports

Dir	Type	Name	Description
In	Variant	Input0	The false input
In	Variant	Input1	The true input
In	Boolean	Select	A selector between the two inputs
Out	Variant	Result	The selected input

Figure 55 Launching a Fleet



The multiplexor, as with all of the Ships we have implemented for this first revision, waits for all three inputs before computing an output. An early completion version could be implemented, but this has consequences for the overall architecture in terms of how sequential operations can then be chained. At the time of this writing it is not clear which option will prevail, making this one of the many questions this Fleet model should help answer.

There is no need for the two data items being presented to the multiplexor data inputs to be of the same type.

13.3.4 Unit: `FIFO`

The FIFO Ship, whose ports are listed in Table 10, is quite simply a buffer for any kinds of Fleet data the programmer desires. It should be noted that this FIFO's depth, unlike those are the micro-architectural level in the switch fabric and such, has functional consequences for the programmer. This is a large part of the decision to model this Ship as an RDL unit rather than an RDL channel, which are intended to model communication, not storage.

Table 10 FIFO Ports

Dir	Type	Name	Description
In	Variant	Input	Input data
Out	Variant	Output	Output data

There is no need for the data stored in this FIFO to be of the same type, as evidenced by the variant input and output ports, which may carry different data types at different times.

13.3.5 Unit: `Rendezvous`

The FIFO Ship, whose ports are listed in Table 11, implements barrier synchronization for Fleet data. Rendezvous Ships will wait for some data, of any type, to be available on each input, and then simultaneously deliver the input data to the output. The actual Verilog unit implementation is parameterized, allowing the Fleet designer to easily create rendezvous of multiple sizes. The combination of token literals and the bitbucket, to which unneeded data may be sent, can be used to create a smaller rendezvous out of a larger one.

Table 11 Rendezvous Ports

Dir	Type	Name	Description
In	Variant[]	Input	Inputs
Out	Variant[]	Output	Outputs

The rendezvous Ship is the primary data synchronization mechanism in Fleet. Note that synchronization also takes place at each box, as the data and instruction must both be present before either may enter the switch fabric.

13.3.6 Unit: `Fetch`

Though not instantiated with the `Ships` unit, we consider fetch a Ship because it has an input for code bag descriptors. The fetch Ship is then responsible for loading the move instructions from the named code bag and inserting them in to the instruction fabric for delivery to the boxes as shown in Figure 54.

Internally the fetch Ship contains a FIFO to queue up the code bags to be loaded, a program counter which turns a series of code bag descriptors into the proper instruction memory accesses, the instruction memory itself, and a decode unit which breaks the raw memory contents into the proper RDL typed messages. In future versions of Fleet this may be implemented programmatically as each piece of this functionality could be implemented by a standard Ship [43].

13.3.7 IO Ships

In addition to the core Ships presented above we have created four IO Ships meant for debugging and demonstration purposes. We have created `Display`, `TokenInput`, `BooleanInput` and `IntegerInput` Ships.

The display is build to display any type of Fleet data presented on its single input. The three input Ships each designed to produce one type of data, based on a button, switch or many switches respectively.

13.4 Examples

In this section we present the three main example Fleet programs we have produced. These are all relatively simple examples meant to show off the power of the tools, and their use rather than Fleet itself. We are currently working on more complex examples and test cases at the same time as we refine the Fleet architecture.

13.4.1 Addition

As an extremely simple test case for the Fleet Program 62 will compute $17+5$ and display the result. This is mostly a test of the literal handling logic and the Fleet launching process (see Section 13.2.4).

Program 62 Addition.fleet

```
1 initial codebag Addition {
2   move (17) -> Adder.Adder;
3   move (5) -> Adder.Addend;
4   move Adder.Sum -> Display.Input;
5 };
```

13.4.2 Accumulator

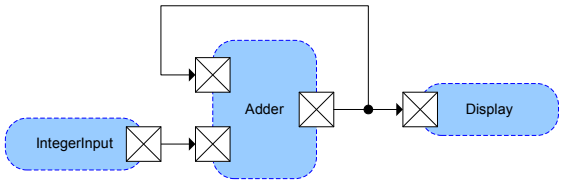
As a first interactive demonstration, we have written a simple accumulator shown in Program 63. This program allows the user to enter a stream of integers and presents a new running sum for each integer entered.

Line 2 specifies that the running sum, which circulates through the adder, should start at 0. Lines 4-5 then set up a pair of standing moves which form the paths shown in Figure 56. This is the power of standing moves in Fleet, they allow the programmer to construct long standing pipelines of Ships to perform streaming calculations without excessive instruction counts.

Program 63 Accumulator.fleet

```
1 initial codebag Accumulate {
2   move (0) -> Adder.Adder;
3
4   move [] IntegerInput.Output -> Adder.
        Addend;
5   move [] Adder.Sum -> Display.Input,
        Adder.Adder;
6 };
```

Figure 56 Accumulator



13.4.3 Counter

Building on the accumulator example, we created a counter example program shown in Program 64. This also provides a pleasing symmetry with the RDL counter example (see Section 7.4. With this example the user simply provides a stream of tokens, each of which will cause the counter to increment by one. In contrast to the accumulator shown above, this means that the user is not providing data, merely synchronization a fact reflected in the increased complexity.

Program 64 Counter.fleet

```
1 initial codebag Addition {
2   move (0) -> Rendezvous.Input[0];
3   move [] (1) -> Adder.Addend;
4
5   move [] Adder.Sum -> Display.Input,
        Rendezvous.Input[0];
6   move [] TokenInput.Output ->
        Rendezvous.Input[1];
7   move [] Rendezvous.Output[0] -> Adder
        .Adder;
8   move [] Rendezvous.Output[1] ->
        BitBucket.Input;
9 };
```

Lines 2-3 set up initial values for the counter, providing the initial count on line 2 and an infinite stream of 1s on line 3, assuring the count will always increase by one. Line 5-8 then set up the connection shown in Figure 57. Note in particular that the rendezvous and bitbucket are used to synchronize the counter loop, which would otherwise run

extremely fast, to the stream of tokens generated by the user pressing a button.

13.5 ArchSim

ArchSim is a Fleet simulation tool written in Java and designed to support a higher level of simulation than our RDL models. In particular it does not have detailed cycle-accurate accounting, as much of the Fleet design has focused on self-timed implementations anyway, nor did it result in concrete hardware implementations. Though it is very useful tool for architectural exploration and design, we hope to replace it with our RDL implementation which is more believable and faster, though less easy to modify.

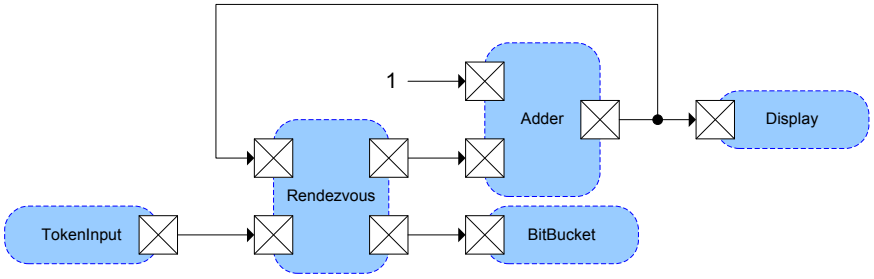
As a way to bridge the transition from one to another, we implemented a language back end for RDLC2 (see Section 10.2) which could generate the relevant input for ArchSim. In particular ArchSim expected a netlist of components, typically Ships, along with a so-called library of these elements linking names to Java classes implementing them. Both of these kinds of data are written in simple XML formats, and correspond neatly to the output of RDLC2 shell (see Section 8.3) and map (see Section 8.4) commands.

While the format and the ArchSim tool are no longer in use, to our knowledge, the simplicity and XML basis of this netlist representation made it very useful for basic RDLC2 testing (see Section 9.6.3). In particular the code generation for ArchSim is a mere 1600 lines of code including extensive comments, making it easy to write and debug. Many of the more complex tests in the automated RDLC2 test suite, which are not geared toward a particular output language, use the ArchSim XML format to avoid testing both code generation and input at the same time.

13.6 Conclusion

Fleet [31, 85, 86, 87] is a novel computer architecture based on the idea that the ISA should be focused on exposing the hardware's abilities to the programmer, instead of hiding them below abstractions. In this section we have briefly presented the details of the Fleet architecture, as well as our RDL model of it, the first ever hardware implementation of Fleet capable of running actual programs. We have shown our implementation of the architectural models, and integrated assembler toolflow both of which were drivers, to a greater or lesser extent of RDLC2 development.

Figure 57 Counter



Chapter 14

P2 & Overlog

In this section we describe our re-implementation of the P2 [69] system and the Overlog declarative networking language on top of RDL, which can be compiled to a gateware implementation.

Nearly all sufficiently large hardware systems, such as those RDL was designed to support, are built on the globally asynchronous, locally synchronous design pattern because it allows components of the system to be constructed and tested independently. Recently, projects like Click [62, 61] and P2 [69, 67, 68, 88], have explored the construction of traditionally monolithic software systems using dataflow components, with a similar communications pattern.

What's more these software systems have admitted a certain performance penalty for the ease of specification and debugging that a dataflow execution model provides. In order to recapture this lost performance, expand the range of applications for these systems and improve the networking functionality available to reconfigurable systems programmers, we have built a compiler which will transform a P2 Overlog specification into a high-performance gateware implementation.

Click was targeted to building router control planes and P2 to build overlay networks (e.g. Chord [83], Narada Mesh, etc) in a succinct and analyzable fashion.

RDL was designed to support large scale multiprocessor computer architecture research, allowing independent researchers to build and assemble complete, accurate gateware simulations, rather than resorting to software, which is typically several orders of magnitude too slow for applications development.

Systems like P2 and Click add value by expressing the system as a composition of simple elements executed using dataflow semantics which eases design and implementation at the cost of overhead. Additionally, the parallelism in the dataflow model is difficult to manage in a microprocessor [36]. This project takes the logical extension of expressing the high parallelism inherent in dataflow models directly in a parallel medium, namely gate level gateware. We show that it is possible to automatically implement complex systems in gateware and obtain a substantial performance benefit by harnessing the implicit parallelism of these systems.

[0]Excerpts from this section have been presented in prior reports and are thanks in part to Andrew Shultz and Nathan Burkhart.

14.1 Background

This project represents the synthesis of several areas of research, namely distributed systems, languages, databases and computer architecture. This section provides background on the various projects which form the basis of our work.

In this section , we present an alternative implementation of the Overlog language and semantics which can be compiled through RDL to Verilog for implementation on an FPGA. Implementing overlay networks in gateware has two direct benefits. Because the gateware implementation is specialized and parallel, it can run orders of magnitude faster than a comparable software system. Second, a gateware overlay network would provide a key component of large scale reconfigurable computing clusters such as the BEE2 [25] used by the RAMP project [90].

14.1.1 P2: Declarative Overlay Networks

In the past several years, research in overlay networks has changed the way distributed systems are designed and implemented. Overlay networks provide many advantages over traditional static networks, in that they enable highly distributed, loosely coupled operation in a robust, conceptually simple manner [69, 83, 55, 78]. However, despite the conceptual clarity that overlays provide their implementation is typically a complex and error prone process.

P2 and Overlog were designed specifically to solve this problem. P2 uses a high level language, called Overlog, to specify the overlay network protocol in a declarative fashion. P2 essentially separates the description of the overlay from its implementation, making it easier to reason about the correctness of the protocol. Furthermore, P2 automates the implementation of the overlay by compiling the declarative description into a dataflow execution. Other projects such as Click have shown the value of dataflow execution models for simplifying the construction of complex systems.

Aside from the complexity problems, overlay networks typically have performance issues and high implementation costs. Because these networks often maintain a large amount of state and a different routing topology on top of the already costly TCP and IP protocols, they tend to have low performance. Additionally, the generality offered by a dataflow model comes with performance costs, especially when serialized to run on a microprocessor, thereby losing most or all of the parallelism.

In order to integrate with the current hot topic applications like firewalls, 10Gbps routers and intrusion detection systems higher performance implementations of overlay networks are required. Worse, the complexity and cost of these implementations often forces constraints on the size of the test bed which can be constructed thereby limiting the reliability of the protocol..

14.1.2 RDL

In RDL all communication is via messages sent over unidirectional, point-to-point channels, where each channel is buffered to allow units to execute decoupled from each other. In a design with small units, like ours, the buffering inherent in the channel model forces delays and increased circuit size. However given the relatively abundance of registers to LUTs in most FPGAs, such as the Xilinx Virtex2Pro, the buffering is not a problem, and the increased latency is less important because the P2 model admits pipelining of operations on tuples.

In this section we will restrict our discussion to FPGA host implementations of Overlog targets. In fact we also spent some time on the code necessary to produce Java host implementations of Overlog targets, but the Java output functionality was temporarily removed from RDLC during a major revision to support this project.

14.1.3 BEE2

The BEE2 [25, 37] is the second generation of the BEE FPGA board originally designed to support in-circuit emulation of radio controllers at the Berkeley Wireless Research Center. The BEE2 was designed to support general purpose supercomputing and DSP applications in addition to the specialized ICE functionality of the BEE.

The BEE2 is also the primary board to be used in the RAMP project. With 5 Xilinx Virtex2Pro 70 FPGAs, each with two PPC405 cores, up to 4GB of DDR2 DRAM and four 10Gbps off-board Infiniband or 10Gbps Ethernet connections, the board includes over 180Gbps off board bandwidth, and 40GB of RAM, enough for even the most demanding applications. Furthermore the bandwidth on and off the board has been carefully balanced to avoid the bottlenecks which often plague such systems.

Because the BEE2 is aimed to be primary RAMP host platform, we have used it as our test platform. Given the speed and implementation density of Overlog designs relative to the BEE2's capacity, it might also provide a useful test platform for overlay networks (see Section 14.2.1).

14.2 Applications

In the previous section we outlined the various projects which are key components of our work. In this section we expand on this to suggest the ways in which our work will contribute back to these projects.

14.2.1 Overlay Networks

Because a parallel gateware implementation of an Overlog program can run orders of magnitude faster than the original software implementation of P2, our works opens up the possibility of running experiments on Overlog programs in fast-time. Furthermore, since a gateware implementation does not time-multiplex a single general processor, more nodes can be packed onto a BEE2 board than can be run on a normal CPU. Time and space compression could allow testing of larger networks than current clusters of CPUs can offer.

In addition, fine grained (clock cycle level) determinism, which is a core part of the RDL model, would allow cycle-accurate repetition of tests, a great boon to those debugging and measuring a large distributed system.

Line speed devices like routers, switches, firewalls and VPN end-points could benefit significantly from the parallelism and speed of these implementations combined with the high level protocol abstraction provided by Overlog. For example, this could allow the design of core-router protocols

using a simple declarative language, and the automatic generation of 1-10Gbps, line rate, implementations of these protocols.

14.2.2 Distributed Debugging Tools

In [80] some of the original P2 authors present a debugging framework for Overlog designs which makes use of reflection to debug Overlog designs using Overlog and the P2 infrastructure. Of course this should be a natural idea given the ease with which such a declarative specification captures the semantics of distributed systems, exactly like the way debugging checks need to be specified. While the reflection and tap architecture presented in [80] is unsuitable for implementation in gateware, we believe that similar concepts will be appropriate for debugging general RDL designs.

The reflected architecture is unsuitable for general RDL first because the meta-information even for a single gateware node could quickly overwhelm the storage available at that node, both in capacity and bandwidth. Even invoking the RDL capability to slow target system time, this would produce generally poor performance. Second, the ability to add and remove dataflow taps which is so simple in software is prohibitively complex, even in reconfigurable hardware. In addition, to support code reuse, RDL designs admit arbitrary hardware units, including unknown state. This would prevent tracing as presented in [80], as the cause and effect relationships between messages is unknown.

However, even with these limitations the RDL model can easily support interposition on channels for monitoring or data injection. Overlog, or a similar language, with support for RDL message types, could provide a concise and understandable mechanism for specifying watch expression, logging and breakpoints with complex distributed triggers. In this case a hardware implementation is a necessity not only for interfacing with the circuit under test, but also for maintaining the data rate which will often well exceed 10Gbps.

14.2.3 Computing Clusters

The major drawback of reconfigurable computing platforms, the BEE2 included, has and continues to be the firmware required to perform computation on these boards. The memory and network remain the two main peripherals to FPGAs, and the two hardest pieces of hardware to interface to. This project aims to alleviate the situation for networking, by bringing a higher level of abstraction, namely Overlog, to bear on the problem.

RDL and RDLC obviate a large portion of the communications complexity by providing a uniform channel abstraction over a variety of implementations. However, the point-to-point model of communication in RDL cannot support dynamic topologies.

Simplified protocols force the use of highly controlled networks to avoid packet loss or corruption, which these protocols cannot cope with. In a 1000 node RAMP system this kind of restriction would be prohibitive. Providing gateware implementations of high level overlay networks could allow their use for general communications, replacing the fragile, unreliable static protocols normally used with robust, adaptable overlays.

14.3 Languages & Compilers

In the previous sections we presented the enabling research and motivating applications for our work. In this section we switch to a more concrete discussion of the code base, including both of the main compilers used in this project.

14.3.1 RDL and RDLC2

The map command may also invoke a series of plugins designed to implement specialized units. This functionality is used to generate e.g. small SRAMs and FIFOs, with uniform semantics but platform specific implementations. This is also used to generate some of the more complex Overlog elements documented in Section 14.4. In truth this borders on allowing behavioral specifications in RDL, however these plugins must still be specialized to each output language family.

14.3.2 Overlog

Overlog is a variant of Datalog designed to manipulate database tuples, implementing distributed inference over a set of relations. An Overlog program consists of a set of relation declarations, where each relation is a materialized table or a stream of tuples, combined with a set of inference rules of the form: `Name Relation1@N(A, B + 1):- Relation2@N(A, B);`.

This rule specifies that a tuple being added to relation Relation2 at node N should result in a tuple being added to Relation1 at node N, with the relevant fields. Notice that both relations 1 and 2 could be materialized tables or tuple streams.

In the original Overlog syntax given in [69], only materialized relations need to be declared and even then they are un-typed. Firstly, because we are generating gateware, which should be efficient, we require that the types of relation fields be declared

ahead of time. Secondly, in order to simplify the planner, and catch a larger portion of errors at compile time, we required tuple streams to be similarly declared. Examples of materialized table and tuple stream declarations for our modified dialect of Overlog are shown in Program 65.

Program 65 Types.olg

```
materialize TName [10]
  for 10 (key Int, Int);
stream SName (Int, Bool, NetAddress);
```

While most of the gateware implemented at the time of this writing can handle un-typed tuples more interesting features like paging materialized table storage out to DDR2 SDRAM to support very large tables would be costly without a certain minimum of type information.

More importantly, in the short term these declarations have allowed us to catch a number of mindless typos and programmer errors at compile type. The dangers of poor type checking in hardware languages are all too real, as Verilog provides almost non-existent and non-standard type checking.

As a final exercise we present the Overlog program snippet in Program 66, which is an extension of one of our tests. It declares two streams, tells the compiler to put a watch of each of them for debugging during simulation, specifies some base facts, and a simple rule for computation.

Program 66 Example.olg

```
stream Stream0(Int, Int);
stream Stream1(Bool);

watch Stream0;
watch Stream1;

Stream1(true);
Stream0(0, 1);
Stream0(2, 3);
Stream1(false);

Stream1@N(A > B) :- Stream0@N(A, B);
```

The expected output of this program is shown below. However the interleaving of the results will differ based on the actual execution timing.

Stream1: <true>

Stream0: <0, 1>

Stream1: <false>

Stream0: <2, 3>

Stream1: <false>

Stream1: <false>

In Section 14.4, we discuss the details of our Overlog planner, and the architecture of the resulting system, but we must also briefly touch on the integration of the Overlog compiler and RDLC2. In Section 14.3.1, we described RDLC as a compiler framework with support for a plugins. These features allow us to specify the Overlog compiler as a chain of program transformations turning an Overlog program into and RDL design which is then turned into a Verilog design. In addition, this chaining could easily support Overlog rule rewriting such as the localization described in [68]. Finally the plugin architecture allows us to specify the static portions of the RDL design using RDL which includes plugin invocations to fill in the implementations based on the Overlog.

This pattern of RDLC2 transformation chains and plugins has also been used to implement a computer architecture compiler with an integrated assembler. We believe it is general enough to support nearly any transformation required.

The Overlog compiler contains four distinct components which are chained together.

1. The front end lexes and parses the input.
2. A resolve transformation performs error checking and variable dereferences.
3. The Overlog planner plugin is invoked by the RDL portion of the compiler to fill the top level P2 unit in with the various units required implement the Overlog program.
4. A series of gateware generator plugins create the actual Verilog unit implementations of the dataflow elements used by the planner.

14.4 System Architecture

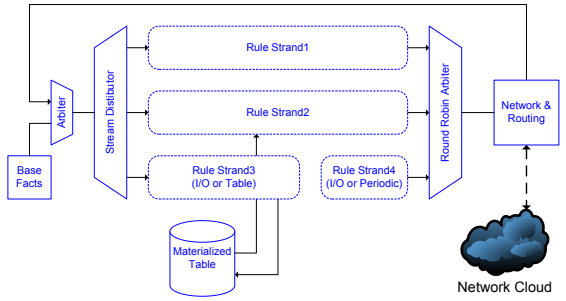
An RDL design is composed of communication units, the lowest level of which are implemented in a host language, in this case Verilog. A P2 system however is composed from a fixed set of elements, assembled to match the input Overlog program.

This section describes the elements we have implemented as RDL units, and the planner transformation which assembles these elements. Shown in Figure 58 below is a complete system including the network, several rule strands, a table and the table of base facts which are used to initialize the Overlog program.

As in P2, our implementation consists of a series of linear dataflow sub-graphs called, roughly one

per Overlog rule. Each strand starts with a triggering event, and ends with a resulting action. Events include updates to tables, reception from the network or timers specified using the special periodic relation.

Figure 58 Node Architecture



Program 67 Types.rdl

```
1 message <IWidth, TIDWidth, NAWidth>
2 mstruct {
3   Data<$IWidth, $TIDWidth, $NAWidth>
        Data;
4   Marker Marker;
5 } Field;
6
7 message <IWidth, TIDWidth, NAWidth>
8 munion {
9   bit<$IWidth> Integer;
10  ::1::NetAddress<$NAWidth> NetAddress;
11  bit<1> Boolean;
12  event Null;
13 } Data;
14
15 message mstruct {
16  bit<1> Start, End;
17 } Marker;
```

14.4.1 Data Representation

This system includes two programming languages and three abstractions of computation, each with its own data model. At the Overlog level, data is presented as materialized relations and tuple streams which are manipulated with the standard relational operators. We write these tuples `<10.1.1.1, 0, true>`.

This is an example of a tuple with a destination network address, followed by an integer and a Boolean field. Notice that this abstraction omits the details of mapping these tuples to actual wires.

RDL units handle tuples as streams of fields, annotated with type, start of tuple and end of tuple signals. The RDL declaration for these messages is shown in Program 67.

From this specification of RDL messages, RDLC automatically generates wires of the specified bit widths, for each field of a union or struct. Union fields are muxed down to a single set of wires for transmission and storage in the channel. In order to support this marshaling, RDLC adds a of tag wires and constants which allow a unit or channel to specify which subtype a union currently holds.

Our representation of tuples was designed with two constraints in mind. First, because the original P2 system uses seconds as the time units, time multiplexing gateware is a profitable way to reduce implementation costs without affecting the functionality of Overlog programs. Even with time units in milli-or micro-seconds, bottlenecks due to the serialization of fields are unlikely, given that most modern FPGA implementations run in excess of 50MHz without difficulty.

The second constraint is on the handling of variable length and un-typed tuples. In addition to components like the network and arbiters, which must handle tuples from widely different relations, our early experiences trying to build distributed databases in Overlog suggested that the ability to store dynamically typed tuples would be a valuable feature. By supporting this kind of processing we can allow future work to build run time programmable tuple processing elements. These elements will be costlier due to lack of typing information, hence the addition of types to Overlog in order to reduce these costs where possible.

As a final note, because the bit widths of the values in our system have been parameterized, it is possible to build smaller or larger systems as needed on a protocol to protocol basis, simply by changing the width of integers or network addresses. In the future we believe a more direct translation between Overlog and RDL types would be helpful both for implementation efficiency and for supporting Overlog as a debugging tool for RDL.

14.4.2 Tables & Storage

Because Overlog is primarily targeted to building overlay networks, materialized tables are slightly different than standard SQL-style tables. In addition to size limits and keys for tuple identity, Overlog includes expiration times for stored tuples.

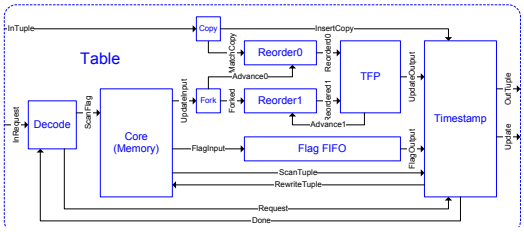
Providing support for all three of these features in gateware was one of the primary sources of implementation complexity for this project. With the high implementation costs of hash-based indexing structures, and the relatively small size of tuples and tables, we chose to implement all table operations as linear scans.

Shown in Figure 59 is the composite table unit, which supports a single input and output. Input requests are represented as an RDL message union of events, which reduces to a set of tag wires with no data.

Program 68 Table Request Message

```
message munion {
        event Scan, Insert, Delete;
} TableRequest;
```

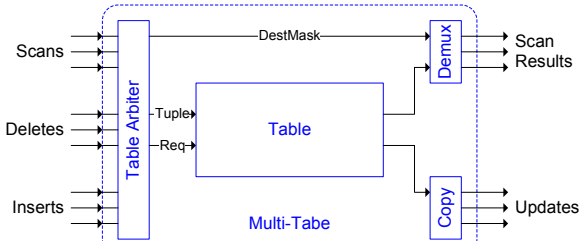
Figure 59 Table Implementation



A scan operation simply iterates over all tuple fields in the table sending them to the output in order. An insert and delete must also perform a join against the input tuple to match keys. In both cases a match implies that the existing tuple should be dropped. Because these operations are implemented as scans, they can in fact perform garbage collection on unused areas of the table memory, by simply rewriting the entire table. Tables can include tuple expiration times, forcing a rewrite on a scan as well in order to avoid outputting a stale tuple.

Because new tuples are always inserted at the end of the table, and the table is garbage collected on each scan, it was a simple matter to implement the drop-oldest semantics for full tables: the oldest tuple will always be the first one in scan order.

Figure 60 Multi-port Table



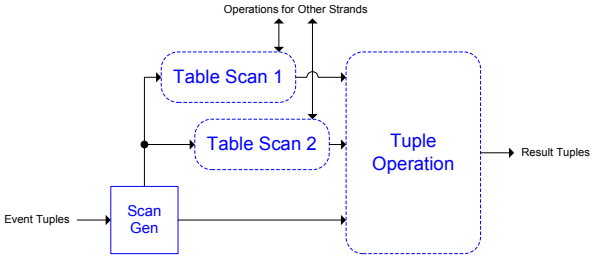
Shown in Figure 60 is the unit used to provide multiple ports to the table, in order to support multiple rules which access a single table. The input stage consists of a round-robin arbiter, modified to allow multiple table scans to proceed in parallel. Since it is common for many rules to use a table in their predicate, this is an important performance optimization.

In our original design system, we intended to pack tuples down to the minimum number of bits, shifting and filling where needed based on types[1]. However, providing support for un-typed tuples made this an expensive proposition, as the implementation would require either a barrel shifter (very expensive in FPGAs) or possibly many cycles per field. Instead each tuple field is stored at a separate address in the table memory.

14.4.3 Rule Strands

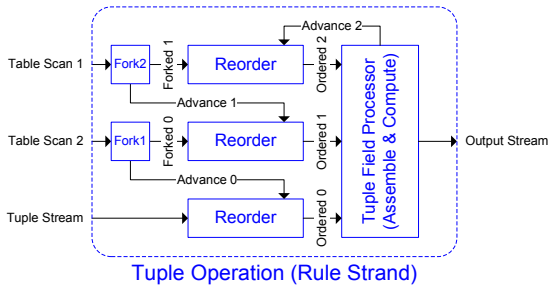
Figure 61 shows a complete rule strand, including the logic for triggering table scans on the arrival of an event, and the tuple operation unit, shown in detail in Figure 62.

Figure 61 Rule Strand



Because each Overlog rule has at most a single event predicate which is joined with many materialized tables, each strand consists of a series of nested scans which are triggered by the arrival of the relevant event. In Overlog an event can be an update to a table, the arrival of a tuple from another node, or a periodic timer event.

Figure 62 Tuple Operation



[1]Not to be confused with RDL packing or marshaling.

Tuples, including the event tuple, and those resulting from scans, are fed into a Tuple Operation unit, which consists of a series of field reordering buffers, chained to implement a nested loops join, and a Tuple Field Processor, which will perform the actual calculations. We describe the Tuple Field Processor in Section 14.4.4.

Figure 63 Field Reorder Buffer

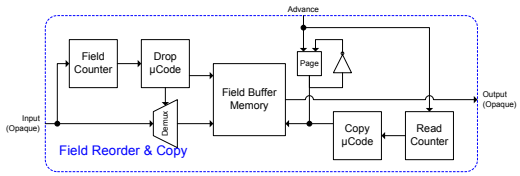


Figure 63 shows the implementation details of the field reorder buffer. These buffers duplicate and drop fields as required to support the calculations and joins specified by the Overlog. For example the rule Result@N() :- periodic@N(A, 10) does not use any of the input fields of the periodic event stream, in which case the reorder buffer for the periodic stream will simply drop all fields. In the rule Result@N(A, A) :- Predicate@N(A, B, C), the field reorder buffer would duplicate the A field, and drop the B and C fields.

The reorder buffers decouple the sequencing of data, which is implied by the output field order, from the operations performed in the Tuple Field Processor. The alternative is direct implementation of the dataflow graph extracted from each rule, with a channel for each variable. However, because tuples will not arrive very close together such a direct implementation would be severely wasteful in FPGA resources to no appreciable benefit.

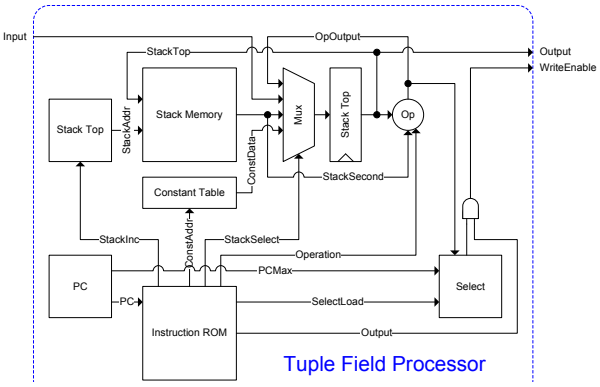
14.4.4 Tuple Field Processor

In order to time share the gateware which performs the computation for each Overlog rule, we built a small stack processor generator. While there must still be one such processor per Overlog rule, this decreases the implementation cost by a factor between 5 and 20, depending on the complexity of the rule.

A tuple field processor has three memories: stack, constant table and instruction ROM. It implements relational join, select, aggregate and computations. Projection is handled in the reorder buffers.

Figure 64 shows a simplified schematic of a tuple field processor. The operations bubble is specialized for the operations required by the Overlog rule the TFP implements. Furthermore, there is no

Figure 64 Tuple Field Processor



support for jump, conditional or loop constructs. Without conditionals, selection is implemented by optionally writing output fields based on prior binary selection conditions that result from both joins and Overlog selection clauses.

Using a processor introduces the possibility of loading new transformations in at run time, by adding a port to write incoming tuples to the instruction memory. This would allow an Overlog node to be dynamically reprogrammed without rerunning the compiler tools. This is less general than full FPGA reconfiguration, since it cannot change the overall dataflow of tuples, however that could be implemented using multiplexor units.

A significant portion of the Overlog compiler code is actually the compiler from Overlog to a custom assembly language for the TFP, and the code which then assembles and links these programs and builds the specialized TFPs to execute them.

Conceptually, the TFP and its assembly language are very similar to the PEL transforms which are embedded in the original P2 system for much the same purpose. However where PEL transform significantly ease the implementation of Overlog in software, the TFP is an efficiency optimization which reduces the size of the generated circuits by almost an order of magnitude for more complex rules.

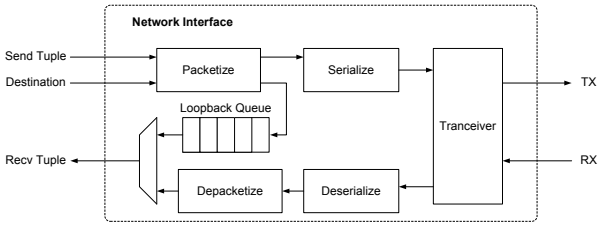
14.4.5 Network Interfacing

The extremely large capacity of modern FPGAs such as the Xilinx Virtex2Pro 70 on the BEE2, enables us to pack many Overlog nodes on each FPGA. This implies that the network infrastructure must span both on-chip and off-chip connections. To this end, we developed a high-bandwidth cross-bar packet switch to connect nodes regardless of their location. As with the components in the nodes themselves, the cross-bar and the network

interfaces were designed and implemented in RDL.

Figure 65 shows a simple schematic of the network interface which connects each node to the network and test infrastructure.

Figure 65 Network Interface



This interface sends packets composed of individual tuples encapsulated inside a tuple containing source and destination information. The tuples come into and out of the interface as a series of fields and are serialized down to an fixed size for transmission through the network.

The switch provides a parameterized number of ports, all fully connected through a cross-bar connection. The choice of a cross-bar enables the highest performance at the cost of increased resource utilization. We have also implemented a "horn and funnel" type switch which uses fewer resources and has lower performance. One of the key points of this network is that the bandwidth is essentially the network width times the number of crossbar ports, lending itself to creating large, high bandwidth networks very easily.

Overlog nodes are not the only end-points on this network. Transceivers and "proxies" can be attached to any port, allowing nodes to communicate from FPGA to FPGA on the same board and on other boards. For our test harness, we use this capability to connect the nodes to a Linux system running on the BEE2 control FPGA in order to inject and collect tuples through C programs..

Finally, the switch instantiates a module which specifies a routing policy for each port. For our test system we use a simple policy that routes based on switch port number with a designated default port. More complicated routing policies such as longest prefix match are also possible.

14.5 Testing

Since our implementation is still in the relatively early phases, we have only run a series of small and synthetic test programs through it. Original we had hoped to run a Chord ring, but the complete Overlog semantics remain more complicated than could be implemented with RDLC2 within a reasonable timeframe (see Section 14.8.3).

14.5.1 Test Overlog Programs

Our tests consist of several example Overlog programs designed to exercise the Overlog compiler, planner, TFP generator and Table implementation. Of course these tests also cover the vast majority of the gateware unit implementations.

Facts: This test was designed simply to display Overlog base facts without processing. This is the bare minimum Overlog program, though it does test portions of the networking gateware, and the majority of the infrastructure code. In addition to a sanity check, this provides absolute minimum implementation costs.

Simple: The simple test consists of a single rule, which fires periodically and increments the sequence number generated by a timer. In addition to the gateware in the Facts program, this includes a periodic timer, and a single Tuple Operation unit, with a single reorder buffer.

Stream: Building on the simple test, this program generates a tuple stream by performing some simple calculations on a periodic tuple, and then runs these tuples through two more rules. This primary motivation for this test was to provide latency and circuit size measurements.

Table: A simple table test, which performs inserts and limited scans over a table which stores a single tuple. This test exhibits a base implementation cost of the table, for size and performance comparisons.

Join: This test adds a larger table, 20 tuples, and performs a join over these to lookup tuples inserted during a specified time range according to sequence numbers from the periodic source.

Aggregate: Performs a series of simple aggregates over a table, including count, min and max.

SimpleNet: A relatively simple network test, which accepts tuples, performs a calculation on them and sends the resulting tuple back to a given address. This was used to test the network interfaces and Linux-based debugging tools.

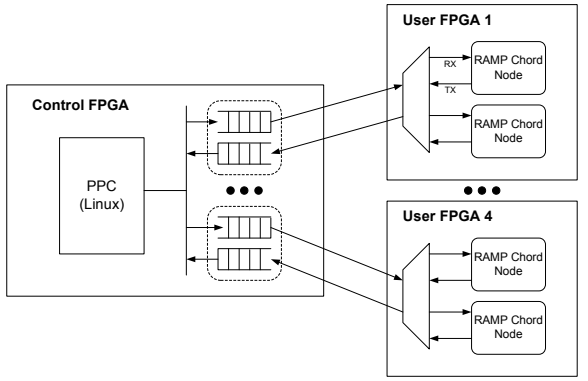
14.5.2 Test Platform

We are currently using the BEE2 as our test platform, primarily because it is used by the RAMP project. The topology of the BEE2 is such that

one of the five FPGAs is designated the "control" FPGA while the remaining four are designated the "user" FPGAs. The control FPGA boots full Debian GNU/Linux on one of the embedded PowerPC 405s in the Virtex2Pro. From this a user can log in to the board, program the user FPGAs and interact with them over high-speed parallel I/Os links.

We have reused infrastructure we originally developed for the RAMP project to connect the tuple network directly to software accessible FIFOs on the control FPGA. The linux kernel on the has drivers which abstract the these FIFOs as either character devices or virtual Ethernet channels. We use the FIFOs as character devices allowing us to write simple C code to inject and read back tuples by reading and writing files. Figure 66 gives a schematic view of this connectivity.

Figure 66 Network Topology



In addition to sending and receiving tuples through this interface, we use the hardware FIFOs to collect statistics and events directly from the hardware. By sharing the infrastructure originally developed for the RAMP project, we have significantly eased the integration of standard Linux software and raw hardware.

Future development with Overlog and RDL will make I/O a first class primitive which can be more easily defined within the language itself. Having I/O defined explicitly in the system description will enable more robust debugging and potentially higher performance communication. By integrating this with the cross-platform capabilities of RDLC, we can also ensure that future projects will be able to share similar communications infrastructure even more easily than we were able to.

14.6 Performance Results

This section presents performance numbers both on the compiler and the gateware that it generates. Given that this project is still in the relatively early stages, these should be considered rough numbers.

14.6.1 Compiler Performance

Shown in Table 12 below are various compiler and simulation performance metrics for the test programs described in Section 14.5.1. Most of these metrics are tied to the high level system design and the conceptual mapping of Overlog onto RDL. Even without a current basis for comparison, these numbers are important as a baseline for our future work.

Table 12 Compiler & Simulation Costs

Test	Mem	Comp.	Load	Sim
Facts	10MB	5.51s	1:07m	1.68s
Simple	13MB	5.65s	1:15m	2.84s
Stream	13MB	15.22s	1:42m	3.43s
Table	15MB	18.96s	1:39m	5.21s
Join	18MB	17.01s	1:28m	5.14s
SimpleNet	11MB	11.70s	1:57m	1.82s

The four numbers, in order, the RDLC and Overlog compiler memory usage and time, minus the 30MB and the time it takes to load all of RDLC and the JVM. At 150K lines of code and 10MB even for a simple Overlog design, the compiler is clearly overly large. See Section 14.8.3 for more information.

The next two metrics are related to the performance of hardware simulations using the industry standard ModelSim SE 6.1e. The load time is measured from simulator invocation to the completion of Verilog compilation, and is a reasonable metric for the code complexity. The simulation time is the amount of real time taken to simulate 100us circuit operation, more than enough time for all of the programs to do useful work. The clock period of the simulation is 1ns, to ease the math, despite the fact that this is unrealistic. Most of the load time is inherent in the simulator we used, especially since it uses a networked licensing scheme. Load times therefore are a relative measure of complexity.

All compilations and simulations in this table were run of an unloaded P4 3GHz with 1GB of RAM.

Aside from the memory hogging inherent in RDLC, most of which is due to inefficiencies in the RDLC2 core, and its Java implementation, these numbers are reasonably promising. We revisit the simulation time in Section 14.8.3 however.

14.6.2 Micro-benchmark Performance

Shown in Table 13 are the results of the Xilinx FPGA Place and Route tools for each test program. These results were produced with Xilinx ISE 8.1 on an AMD Opteron 146 with 3 GB of memory. For these we used the included XST synthesis tool, rather than the more powerful Synplify Pro because of issues with IP licensing for Xilinx specific cores.

Table 13 Hardware Statistics

Test	#LUT	#FF	Clock	Time
Facts	468	450	175MHz	1m 35s
Simple	1155	867	172MHz	2m 1s
Stream	2260	1546	173MHz	3m 47s
Table	2303	1655	173MHz	3m 25s
Join	2606	1837	177MHz	3m 50s
SimpleNet	980	806	176MHz	1m 55s

The LUT and Flip-Flop counts are presented for two reasons, first, they impose a hard limit on the number of nodes which can be implemented in an FPGA. In addition they affect the PAR tool run time and clock frequency the synthesized circuits can run at.

Given that the largest tests implemented roughly a single Overlog rule, a Virtex2Pro 70 could hold roughly a single Chord node. However, we believe that these hardware costs could be significantly reduced with better compiler optimizations.

The most impressive numbers in Table 13 are undoubtedly the clock frequencies. A 100MHz design on a Virtex2Pro is fairly standard, and not too difficult, but normally anything over this must be hand optimized for performance. From the fact that RDL designs with small units generally balance LUT and Flip-Flop usage, it is clear that these designs are highly pipelined, with increases their operation frequency into a range unheard of for automatically generated designs.

As impressive as the raw numbers in Table 13 is the simple fact that P2 takes 100ms to respond to a query, whereas our gateware implementation will typically respond in maybe 100 clock cycles, which at 100MHz results in a 1us turn around on input queries, a very impressive result for a such a high level input language as Overlog. Of course the price for this performance includes the cost of owning and FPGA board, which many researchers do not.

However the biggest price, is undoubtedly the cost of recompiling a system through the FPGA synthesis tools. While the runtimes for these projects are reasonable, many larger BEE2 applications have been known to take in excess of 12 hours, implying that progress in this area will be required to make gateware Overlog implementations as flexible as their software counterparts.

14.7 Conclusion

By and large, this project marks a considerable success and the confluence of several research projects. By themselves Overlog, RDL and the BEE2 are all interesting, but combined they promise to open up both new research avenues and new application areas. Despite the successful execution of this project, there exists a significant amount of work to be done, as outlined in Section 14.8. In the remainder of this section we detail the lessons from our implementation efforts, and discuss their impact.

Many of the implementation decisions in this section relate to the running time and space of our design. When switching from software to hardware, $O(1)$ becomes $O(n)$ because operations reflect their per-bit cost in the absence of a fixed bit width CPU. This caused us consternation during the design process, as we tried to optimize all of our operations, with little regard to relative run time of hardware and software. What's efficient in hardware is not the same as what is efficient in software and a project, like this, which spans the two can be tricky to design. In the end we took the view that a functional result was the primary goal and speed could wait. We feel justified in this, as it was often unclear what was feasible and what was not, and with 150 thousand lines of code in the various compilers, it has been a significant effort to get to this point. Furthermore the hardware implementation by virtue of its specialization is faster than most software could ever hope to be, even without heavy optimization.

In the end the biggest drawback of the current implementation of the Overlog compiler and language relates to its inability to handle system level I/O. While our test platform provides a clean link to software which can inject and read back tuples, the process of making this connection to an I/O block needs to be automated in the compiler. We believe this will prove to be one of the main requirements for useful systems written in Overlog, just as the ability to generate non-channel connections was key to making RDL a useful language for this project.

The best news at the conclusion of this project is the relative ease with which it was completed. Normally any hardware design this large might take

a man-year or more. However we implemented it in 3 months, including 2.5 man-months of coding for RDLC2, and the Fleet (see Section 13) compiler, an extension to RDLC2 similar to the Overlog compiler, which provided good debugging tests for RDLC2. The Overlog compiler and elements themselves took about 3 weeks and we were able to implement the complete system in about 6 man-weeks.

Furthermore, those Verilog modules which are generated by Java plugins for RDLC are very powerful, allowing the Tuple Field Processor and Reorder units described in Section 14.4 to be built from scratch in about 2 days total, with another half day of debugging. Considering that these units amount to a small data cache and processor, along with an assembler and processor builder, this is an almost unheard of time frame.

We believe the modularity of RDL, combined with the high-level semantics of Overlog contributed significantly to the ease of development. For example, the testing was quite laborious until we implemented the Base Facts unit to supply raw tuples specified in an Overlog program to a P2 system at startup. What's more it took only a few hours to add all of the language and compiler support for this feature.

Overall the success of these compiler tools in assisting their own development suggests that they are most definitely useful, and we look forward to building real applications with them. This success has also been a large part of our interest in applying these tools to architecture debugging for the RAMP project.

14.8 Future Work

A significant fraction of the time to complete this project was simply getting to the point where sufficiently complicated RDL could be compiled to gateware. In the end, the quality of implementation of the Overlog compiler has been sacrificed somewhat for speed of development: there's a definite lack of flexibility in the RDLC2 framework and supported Overlog constructs.

For example a better framework for the planner would easily allow us to implement Chord, as outlined below, as well as providing compile time optimizations like constant propagation and expression simplification.

14.8.1 Chord in Hardware

In the relatively short term we hope to be able to boot a chord ring in gateware. By adding type declarations, and some minor lexical and syntactic changes, we were easily able to compile the Chord specification used for P2 testing, but it cannot be transformed into RDL with the current version of RDLC2 and the Overlog planner.

The first big problem here is the use of null or out-of-band values. We did not implement support for these well enough, and changing this would have required rewriting large portions of Java that generates Verilog. Furthermore compiling Chord also exposed a latent bug in our handling of nested table scans for rules with multiple materialized table inputs and again the fix for this would have involved painful surgery on Verilog. We will discuss these problems in Section 14.8.3.

14.8.2 Java Implementation

Early in this project, while we were using RDLC1, which has Java output support, we actually did a fair amount of work on a Java implementation to match the gateware. Because RDLC is designed generate code for software and hardware hosts equally easily, this would open up interesting possibilities for research. Firstly, this would provide option to split the implementation between hardware and software. Second because an RDL software system is essentially a highly parallel program with a user level scheduler, this would promote a whole range of systems research from schedulers to protection and IPC.

The main different would be a lack of TFP in software, as having access to the RDLC Java code generation facilities would have allowed us to hard code the tuple operations without the need for a TFP or PEL transform.

14.8.3 RDLC3

Many of the problems we encountered were related to shortcomings of RDLC and the compiler framework. While the second generation code base greatly increased our capabilities the actual compiler code itself is still rather messy, making changes to plugin compilers, like Overlog, difficult.

Our biggest problem was the immaturity of the library of gateware units upon which we could draw. Tools like the Xilinx Core Generator already exist in this area but are vendor specific, do not fit the RDL model and are tend to be both buggy and closed source. One of the features on the short list for RDLC3 (see Section 16.4.1), and key to our implementation of Chord is better integration of generated units into the compiler either in the form of macro-replacement, language fragments [16] or at least a better Verilog code generator.

Expanding the unit library to include a DRAM

controllers, as opposed to just the SRAM generators which we built, would allow us to build a paging mechanism to store of large relations in the DDR2 DRAM on the BEE2. DRAM controllers as mentioned in Section 14.1.3, remain one of the most time consuming pieces of firmware and yet they will be required to produce large scale Overlog systems which go beyond simple overlay networks.

14.8.4 Debugging Tools & Features

From the 2-3min turn around time on debugging even our micro-benchmarks, it became clear that we need better performance out of the simulation environment for those situations where an FPGA is a poor test platform.

In addition, we faced a significant number of crash bugs in ModelSim during the course of RDLC2 development. Some were alleviated by an upgrade, but some have been documented by the authors for up to two years now without a forthcoming fix.

In addition to finding a better simulation environment, we believe the by developing on the ideas in [80] and Section 14.2.2 we could more easily debug Overlog, the compiler itself and general RDL designs. As part of RDLC3 we plan to integrate the Overlog compiler with the forthcoming RDL debugging framework to support debugging not just of Overlog designs, but of all RDL designs.

We believe these enhancements, experience and maturity in our tools with lead to their use in real systems in short order, as they provide much needed functionality.

Chapter 15

RAMP Blue

The goal of the RAMP Blue project [79, 63, 84, 64] was to implement a large scale multicore system on multiple BEE2 [25, 37] FPGA boards. RAMP Blue has been highly successful at this goal, resulting in several demonstrations of the system running the NAS Parallel Benchmarks.

However the major goal of the RAMP project has been to create simulationssimulators for such systems, something which the original RAMP Blue work [79] did not really attempt. In a large part this was because RAMP Blue and RDLC2 were being co-developed in such a time frame as to prevent their tight integration. Later work [84] ported RAMP Blue almost completely in to RDL as shown in Figure 67.

As it stands now the RAMP Blue project is somewhat complete with all of the source code, and build instructions, available from [9]. However, RAMP Blue is still an implementation of multicore system without reference to the kinds of simulation RDL was designed to support. Thus while RAMP Blue and RDL have been brought together, a time consuming project thanks to CAD tool issues, there remains a somewhat elusive research goal of converting RAMP Blue from an implementation to a simulation. Along with this there are performance issues attendant upon the design of the RAMP Blue network and abstraction issues, as shown by the fact that the inter-FPGA connections are not yet properly abstracted as RDL channels in Figure 67.

15.1 Overhead

As with any CAD tool or language, there is a tendency to assume that the power and flexibility come at a price of some overhead in design area or running time, and efforts to quantify this with RAMP Blue abounded. However, RDL is a language meant to capture a certain class of system, primarily those designed to fit within RDF. There is no overhead inherent in RDL, because RDLC does not perform any encapsulation or optimization: it is a mechanical, non-expanding transformation of a user specified design.

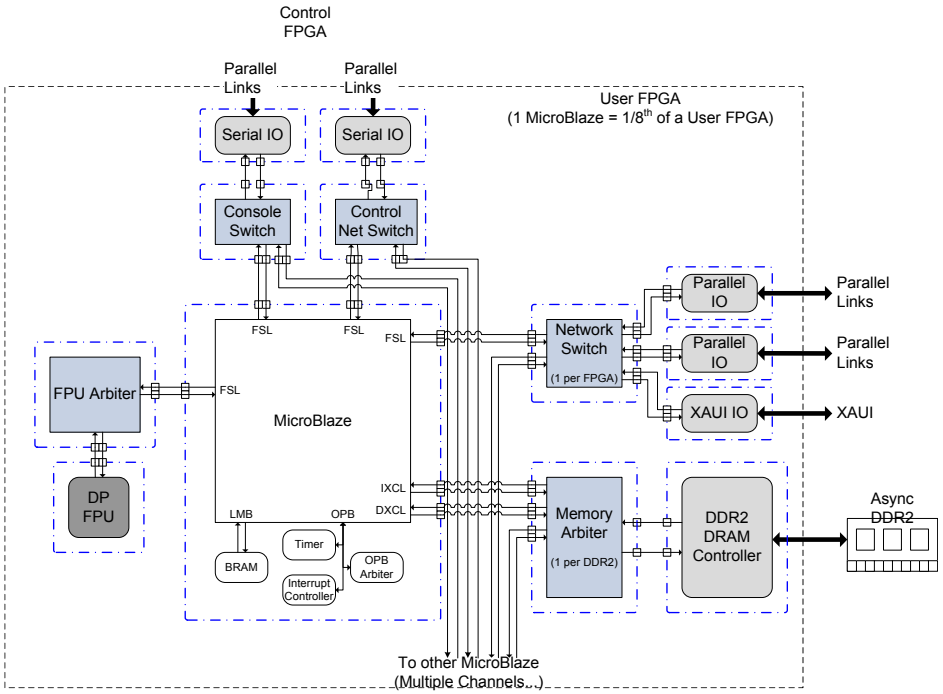
It may be the case that two systems: one specified using RDL and one specified without RDL are superficially similar, but the RDL system is larger. In this case there is a temptation to claim that RDL introduced the overhead, but the truth is that if the system specified in RDL is larger, then it is a more complex system.

Given the flexibility of RDL, and the compiler plugin system it is entirely possible to have highly inefficient designs specified in RDL. Our point in this section is that the resource cost or running time of the resulting system is entirely a function of the design itself, or perhaps the plugins used, rather than a result of a language flaw.

While porting the RAMP Blue [79, 63, 64] to RDL [84], it was discovered that the version of RAMP Blue described in RDL was larger, by 15%, and slower by up to 50%. This comes directly from two contradicting requirements: modifying the original Verilog as little as possible, and conforming to RDF. Modifying the original Verilog would make code updates from the ongoing RAMP Blue project hard to integrate. Violating the assumptions of RDF would greatly decrease the value of the resulting system. In the end, allowing these inefficiencies temporarily, until the RAMP Blue code is stable, was the right decision.

In a technical sense the problem comes from the duplication of handshaking and buffering logic. The original RAMP Blue code was designed not to require, or allow, buffering along where it was broken into units[84], and therefore contains all its own buffering and handshaking logic. In order to conform to RDF, the RAMP Blue in RDL design added buffering between units which was not, strictly speaking, necessary. However, moving forward, this buffering becomes essential as RAMP Blue is used as a performance simulator, and expanded with units which rely on the delay insensitivity requirement of RDF. Once the RAMP Blue is stable, it should be possible to fork the HDL development, and create an RDF conforming version

Figure 67 RAMP Blue in RDL



with comparable size and speed, by removing the unnecessary logic inside the units.

During the evaluation of RAMP Blue in RDL [84] covers the resources and performance of RAMP Blue using three different link generators, each with a fixed channel timing model (see Table 4). We will mention a fourth link generator and timing model which we call "TrueWireLink", with no buffering or handshaking capabilities, and thus requiring no resources. Circuit diagrams for all four basic links are shown in Figure 33.

RDF does restrict a design to at least "DualRegisterLink" or "RegisterLink" as it does not allow 0 latency channels (see Section 3.5) though of course RDL does not have this limit (see Section 3.6). One might consider this overhead, as a cheaper, but functionally identical implementation, may be possible by avoiding RDF. However, it is suspected, though not yet proven, that no such "cheaper" implementation will ever be possible.

Given the possible existence of the "TrueWireLink", RDL cannot introduce overhead, as by using that link, the resulting design will use no more resources than the resources which are part of the units. While this would allow debugging and structural analysis of the resulting design, few of the RDL benefits and none of the RDF benefits will be available in this degenerate use case. Using a more complex link will give more benefits, but cost more.

What is vitally important is that this decision is up to the designer, and RDLC must blindly implement the designers choice, making the use of the term "overhead" rather hard to justify. In short, RDL is closer to a system macro language, a system of generators and syntactic sugar, rather than a complete abstraction of hardware which would necessarily introduce the overhead of abstraction mismatch.

15.2 Conclusion

RAMP Blue, being the largest and most widely known design for which RDL has been employed thus far, has been an invaluable design driver and for exploring RDL as presented in Section 15.1. We have only briefly discussed RAMP Blue as its development has been adequately documented elsewhere [79, 63, 84, 64]. There remains a significant amount of work to be done yet to realize the complete goal of using RAMP Blue as a simulation, but even in its current form it is a useful system both for multicore research and as a driver for RAMP.

Chapter 16

Conclusion

The RAMP [9, 90] project is a multi-university collaboration developing infrastructure to support high-speed simulation of large scale, massively parallel multiprocessor systems using FPGA platforms. The primary goal of RDF is to support the RAMP project, and in turn manycore research (see section 2) and the original RAMP proposals [90, 75].

There are three major challenges facing CPU designers [74]. First, power has become more expensive than chip area, meaning that while we can easily add more transistors to a design, we cannot afford to use them all at once, giving us a "power wall." Second, memory bandwidth is plentiful, but DRAM latency has been getting much worse, relative to the speed of processor cores meaning we can no longer access more than a small fraction of memory at once, giving us a "memory wall." Third, CPU designers have become increasingly adept at exploiting the instruction level concurrency available in a conventional sequential program, but the inherent data dependencies mean we cannot execute everything concurrently, giving us an "ILP wall." Together, these three walls constitute a significant problem for computer architecture, and therefore an opportunity for both academic research and industry to bring parallel processors in to play.

RAMP seeks to provide the tools necessary to allow not only architectural, but operating system, application and algorithm research, by constructing inexpensive, relatively high performance architectural simulations. FPGAs represent a new direction for simulation, promising reasonable performance, price and flexibility, by bringing the efficiency of hardware to bear on the problem of simulating hardware. By constructing a community around shared simulation platforms, designs and tools, the RAMP project will ensure that computer science researchers can cooperate, and hang together.

The U.C. Berkeley RAMP group has been working primarily on so-called structural model simulators wherein the simulation mirrors the desired architecture, making them slightly more believable, and greatly simplifying the transition from simulation to implementation. We have approached this problem by developing a decoupled machine model and design discipline (RDF) together with an accompanying language (RDL) and compiler (RDLC) to automate the difficult task of providing cycle-accurate simulation of distributed communicating components. In this thesis, we have described the goals and models of RDF, its implementation in RDL, several related tools and three applications of varying complexity.

RDF is a combination of abstract models, and design techniques based on the research and collaboration goals of the RAMP project. In contrast, RDL has grown to be a more general hierarchical system netlisting language based on the models of RDF, but not all of the restrictions. Originally for ease of implementation reasons, RDL admits timing coupling between units and latency sensitivity, both of which have proven useful in some of the first applications of RDL (see Sections 13, 14 and 15).

RDF and RDL are designed to support cycle-accurate simulation of detailed parameterized machine models and rapid functional-only emulation. They embrace both hardware and software for co-simulation, debugging and experimental visibility, particularly during the development of complex simulation implementations or evaluation of new architectural features. In addition the toolflow, including a powerful plugin framework for expansion helps to hide changes in the underlying implementation from the user as much as possible. This helps groups with different hardware and software configurations, and even those with little to no FPGA experience, to share designs, reuse components and validate experimental results.

RDF is structured around loosely coupled units, implemented in a variety of technologies and languages, communicating with latency-insensitive protocols over well-defined channels. This thesis has documented the specifics of this framework in-

cluding the specification of interfaces that connect units, and the functional semantics of the communication channels. In this thesis we have covered the technical, theoretical and motivational details of the RAMP model, description language and compiler.

16.1 Lessons Learned

Many of the technical lessons learned during this project are already well incorporated in to this thesis. In particular designing the RDF models has been an ongoing project, as we bring in more and larger applications and as we interact with RAMP groups from other universities. Though difficult at times the diversity of the RAMP project provided us with several rounds of insight and revision necessary to create the models presented here.

At the beginning of this project, we did not foresee that some groups would want to build behavioral model simulations in hardware, as our early work was founded on the premise that implementation HDL could be easily converted to simulation models in the form of units. This assumption hasn't entirely proven false, but it has proven less true than expected as evidenced by the Hasim [38] and ProtoFlex [28] projects, both of which started shortly after RDL and are building hybrid structural model and behavioral model simulators. Furthermore, seemingly minor details like the inflexibility of our inside edge (see Section 3.2) interface or the lack a certain kinds of parameterization at the language level have proven troublesome for some potential users. In response to this we have already begun working on incorporating these kinds of changes in to the next generation of RDL (see Section 16.4.1).

As with any new tool, and particularly a computer language RDL has faced a strong uphill battle for adoption. The above technical barriers are of course no small part of this, but some missing features or misdesign adventures are to be expected and valuable in their way. What we did not expect was the level of resistance to a new tool set on the basis of its perceived immaturity or failure to solve all possible problems.

Many potential users inside and outside the RAMP project have remaind reluctant to use RDL not only because it is missing some minor feature, but because they have no wish to spend time learning a new tool until it solves many, if not all, of their problems. The time spent learning, and hopefully eventually contributing, to a shared tool is by default perceived, correctly or incorrectly, as being higher than the return for individual researchers. Unfortunately the narrowness of the initial application pool used to guide the design of RDLC2 has made it difficult to quickly accommodate the needs of these researchers, thereby worsening this perception.

On this basis, we believe that any tool, like RDL should be designed from the start with the widest possible range of applications and unit tests (see Section 9.6) in mind, though still with a focus on key applications. Compiler level data structures, and simple choices about parser generators have turned out to have an impact well beyond our expectations. Not only do these seemingly uninteresting, to the computer architect hoping to use RDL, design decisions affect the compiler at the most fundamental level, but with the time pressure attendant on a research project and the coding style that fosters, they have proven extremely difficult to change later.

Since its release we believe that both the users, and those who have avoided RDL have given us a significant amount of feedback on technical details which we present very briefly in Section 16.4. However we have also received feedback from students eager to work with our tools, leading us to believe that with time and work RDL will definitely find widespread use. On this note there is a rule of thumb which we have come to appreciate: "If a tool cannot be learned by a busy professor or a complete novice in an afternoon it is unlikely to be useful." We believe that a key component of any future work should involve the creation of tutorials, full scale RAMP system demonstrations and perhaps even course type material in order to lower the perceived barriers to use.

In short we have learned, not only a number of technical lessons, but quite a bit about what users expect of a tool like RDL and what we are capable of delivering. Users expect more than we were prepared for with RDLC2, but the breadth of the RAMP project and the constructive criticism from these potential users has given us a tremendous insight in to where RDL can go next.

16.2 Related Work

At a superficial level it is tempting to compare this work to other HDLs and modelling languages, particularly Liberty [70] and BlueSpec [56]. While the parameterization and interface specification components of RDL are quite similar to these languages, this is a consequence of the inevitable needs of any system description language. Where RDL differs heavily is in its native abstraction of timing and mapping. In particular we know of no existing

tool which is designed to automate the process of creating cycle-accurate hardware implementations of hardware simulators. Furthermore, neither Liberty nor BlueSpec, incorporates the same separation of target and host, allowing the seamless mapping of a design to different hosts, a major prerequisite for the RAMP project.

Of course the basic dataflow semantics underling the RDL target model, as well as the concept of a distributed event simulator are well studied. Projects like Ptolmey [66] at U.C. Berkeley, Kahn Process Networks, Petri Nets, CSP [52], Click [62], P2 [69] and countless others come immediately to mind. Many skilled writers have covered these topics elsewhere, and we gratefully build on their work.

Finally the example applications for RDL are interesting research projects in and of themselves, as mentioned in the relevant sections (see Sections 13, 14 and 15).

16.3 Project Status

Two major versions of RDLC were released in the first half of 2006, both with internal Javadocs and several examples. RDL and RDLC2 are stable, with working examples, and are ready for research use, as demonstrated by RAMP Blue (see Section 15).

Timing-accurate simulations have been implemented for RDLC2, and tested but remain unreleased as there are no links with useful timing parameters implemented as of yet. We are working to correct this deficiency at the time of writing, in the form of a `FIFOLink` plugin supporting a complete timing model. It is unfortunate that is has not been released before now, but man hours have been an extremely scarce resource on this project, and users are understandably reluctant to render assistance given how busy they are with their own projects.

RDLC2, the counter example (see Section 7.4) and the Javadocs are all available from [9], as are all of the example applications. Alternatively, the author will be happy to respond to e-mail requests for these items if the website is unavailable for any reason. We recommend the current release, version 2.2007.8.13 as of this writing.

16.4 Future Work

RDLC2 was created in order to address the need for greater parameterization and more complex syntactic constructs in our example applications. In particular, parameterization and support for hierarchical platforms were infeasible within the RDLC1 code base.

Moving forward there are new RAMP subprojects starting at U.C. Berkeley which the author is interested in. It is likely that these will form the new corpus of applications for which RDLC3 will be built.

16.4.1 RCF & RDLC3

A compiler rewrite to create RDLC3 has been planned nearly since the release of RDLC2, mostly in response to fragile nature of the parameter inference algorithm and code generation packages. In particular the reliance of RDLC2 on the base Java abstract data type libraries resulted in short sighted decisions in the higher ordered data structures and algorithms. This in turn resulted in brittle code, which though easy enough to expand so far, has reached its limits in terms of genuinely new language features. To this end we began the development of the RCF libraries with the RADTools project (see Section 11), and we have continued to develop them as a side project.

RDLC3 is designed to simplify the compiler internals, merging and then better exposing many of the data structures to plugins and outside tools. With the compiler internal upgrades, we hope to add support for expressions over parameters, and better output code generation in the form of more complete abstractions (see Section 10.2). Types for parameters, including type bounds, Turing-complete parameterization, generation support and simple arithmetic expressions are all on the expected list of additions. Furthermore, we hope to improve the parameter inference algorithm, in particular error reporting and interaction with arrays is quite poor as consequence of deadlines cutting in to design time, and insufficient negative test cases. It should also be easy to add parameter arrays and declaration arrays whose dimensionality is parameterized or inferred, allowing a parameter to specify when to use a 2D vs. a 3D structure.

The addition of type bounds for RDL parameters, suggests the possible addition of unit and platform polymorphism similar to what is available in Java, C++ and Liberty [70]. It is not yet clear if the use of port structures for interface specification will provide enough control or whether some inheritance hierarchy will be needed.

Of course the final implementation and integration of the automatic RDL mapping algorithm (see Section 12) will be a key part of RDLC3. This integration is dependent on a cleaner abstraction of the RDL AST than RDLC2 can currently provide. In particular the generation of the integer programming constraints from the RDLC2 AST would

be complicated and error prone at best, simply because the access needed was not envisioned are part of the design.

We also intend to shift from the JFlex [60] and CUP [54] parser generators to ANTLR [73], which besides using a better parsing algorithm should allow RDLC3 to generate more helpful error messages, a perennial sore spot for users. The more powerful parser should also allow us to restructure the language syntax to reduce the reliance on keywords, improve the specification of types, and general reduce the verbosity of RDL. Trivial changes in this regard can make a programming language of any kind significantly more accessible to novices, whose focus is generally on the lexical structure and syntax rather than the language semantics. Further, given the the number of special purpose keywords (see Section 5.4) in RDL2, we believe an annotation mechanism integrated with the plugin framework (see Section 10) may be a more flexible and simpler choice.

In addition to language level changes, we believe that there are abstraction changes necessary to grow to truly large designs. In particular we have found the lack of a platform level I/O abstraction to be limiting when dealing with a wide variety platforms. Because RDL does not capture platform I/O it automatically manage it, something novice hardware designers and even some experts will assuredly require. We believe that adding the concept of platform level firmware to RDL, including the language and compiler structures to support it will be relatively easy and go a long way to fixing this problem. Furthermore, this should replace the awkward language and platform qualified plugin system (see Section 6.5.2).

Since the initial design of the channel model (see Section 3.3) it has become clear to us that there are other channel models which would be useful. In particular shift-register channels, in contrast to credit-based flow control channels, would not require handshaking and could be used to implement many designs much more efficiently. Expanding on this, several users have requested the ability to specify different variants of the inside edge interface (see Section 3.2), all supporting the same concepts but with slightly different signaling conventions. Integration of existing code, in particular reducing the workload described in [84], would be simplified by features as simple as the option to specify that certain inside edge signals are active low.

We believe a generalized inside edge interface could allow RDLC3 to support several novel features with little cost. Ideas like allowing a unit implementation to pipeline the simulation of target cycles, or a link which can undo or redo the sending of messages within the current target cycle are good examples. More interesting is the idea of making space-time tradeoffs by, for example, delivering the elements of an array message at a rate of one per host cycle to a pipelined unit implementation. A variant of this called implementation multithreading (see Section 16.4.2) is likely to be critical in larger systems. Taking this to an extreme, ideas like message cancellations which flow opposite normal messages could lead to very large performance improvements [35, 34] for certain target systems.

We have planned from the beginning but never had the time to implement support for the simulation of globally asynchronous, locally synchronous (GALS) designs, and other such designs without a single target clock. In particular we have worked out, but not yet implemented many of the details to support units to have dependent clocks related by simple rational divisors.

It should be clear from the above ideas that we have successfully identified many shortcomings in RDLC2. Though RDLC2 successfully and greatly expanded the range of systems for which RDL could be used, we expect RDLC3 will make a significant impact by finally delivering on the complete feature set needed by the majority of projects, in a clean, simple and unified manner.

16.4.2 Implementation Multithreading

Because the RAMP project hopes to scale to thousands of processor cores without thousands of FPGAs, it is clear that simple structural models will not be sufficient. To this end, we have been exploring a concept we call implementation multithreading, whereby a single unit implementation simulates multiple instances in the target system. By time multiplexing the unit implementation, we can allow a researcher to trade time, of which there is an infinite amount, for FPGA or CPU space, which is extremely expensive.

16.4.3 Debugging & RADTools

Debugging any concurrent system is well know to be difficult, if not impossible in real world conditions thanks to the non-determinism underlying the synchronization failures we most often would wish to debug. Because a properly implemented[1] RDL simulation is completely deterministic and repeatable it is possible to expose these bugs. Better

[1] By this we mean that unit implementations are deterministic.

yet, because an RDL simulation will run at hardware speeds, it should be possible to simulate long enough to trigger even the most troublesome failures. Therefore we believe that significant progress in concurrent hardware debugging and testing can be made based on RDL.

RDL provides a clean system level abstraction, meaning it should be possible to build system debugging tools which automatically insert themselves in to a design. A design loader front end based on RADTools (see Section 11) combined with a series of RDL plugins to insert debugging logic and connect it to a software front end as shown in Figure 68, would open up a myriad of possibilities. Best yet, the entire communication between the front end and back end can be abstracted as an RDL link, thereby automating the process of getting data out to the debugger in a flexible manner.

Taking this a step further, the integration with RADTools could form the basis of an automated experimental control facility. In particular the RAMP projects will likely be creating many processor based designs, requiring some tools for loading executables in to the processors, and getting results back. Such an application test server is a natural generalization of the distributed system management facilities in RADTools.

Taking the concept of debugging and RDL even further, we believe that an active debugging framework could be built on the same style of primitives use by the P2 (see Section 14) and Fleet (see Section 13) projects. We envision using the P2 style declarative rules to specify these operations as shown in Figure 69, and programming them in to statically generated debugging hardware automatically inserted in to a design. In particular, the ability to program message processing and injection in to a debugging network could allow interactive debugging, at-speed data processing and even implementation hot fixes.

16.4.4 Non-RAMP Uses

There have been several potential non-RAMP users of RDL since the release of RDLC2. In particular RDL could be used as an implementation language for DSP designs given the right libraries, and for ASIC prototyping with the addition of link generators optimized for ASIC implementation. Building on this, the ability to describe DSP designs compactly in RDL could lead to power estimation tools for RDL designs being themselves written in RDL and integrated with the simulation, similar to the debugging tools described above. We have not yet focused on these applications, given the breadth and complexity already offered by the RAMP project.

16.4.5 Libraries

With any new computer language, it is often not the language which is seen as most useful but the standard libraries which it enables. Researchers have been reluctant to develop RDL units and libraries thus far, making it clear that this is a prerequisite of RDL adoption, not an after effect in their minds. To this end there are several common libraries which could, and should be developed using RDL.

NoC designs are increasingly prevalent in all kinds of digital logic systems. This fact, coupled with their strong commonalities and simple structure suggests that a relatively small library of units and generators could support a wide range of NoC designs and research. Furthermore, these implementations could be used to build things like host level networks to abstract inter-platform connections (see Section 4.4.1).

Of course the RAMP projects, all being centered on multicore processor design have similar needs for processor cores, networks and memory systems. As with NoCs, we envision that the regularity of caches and memory systems should enable the creation of a library of units and generators allowing a researcher to easily model any cache desired. Processor are less regular than memory, but techniques based on instruction set translation [30] and automatic processor generation [76, 29, 49, 91, 50] may eventually allow a similar library of processor cores. Of course designs like RAMP Blue (see Section 15) should eventually be built on these libraries to allow large scale system simulation and experimentation, the main stated goal of the RAMP project.

The platform independence, and thus the adoption of RDL, is tied to the list of link generator plugins (see Section 10.4.1) which have been developed. Of course there are several links missing from the current list, particularly some Ethernet or TCP/IP link, and software-based links using sockets, the Java Native Interface [72] and Verilog VPI [82] to make good on the promise of hardware and software co-simulation. Taking this a step further, the development of a standard set of unit tests and even link implementation pieces could go a long way to expanding the list of implemented link generators.

Figure 68 Debugging

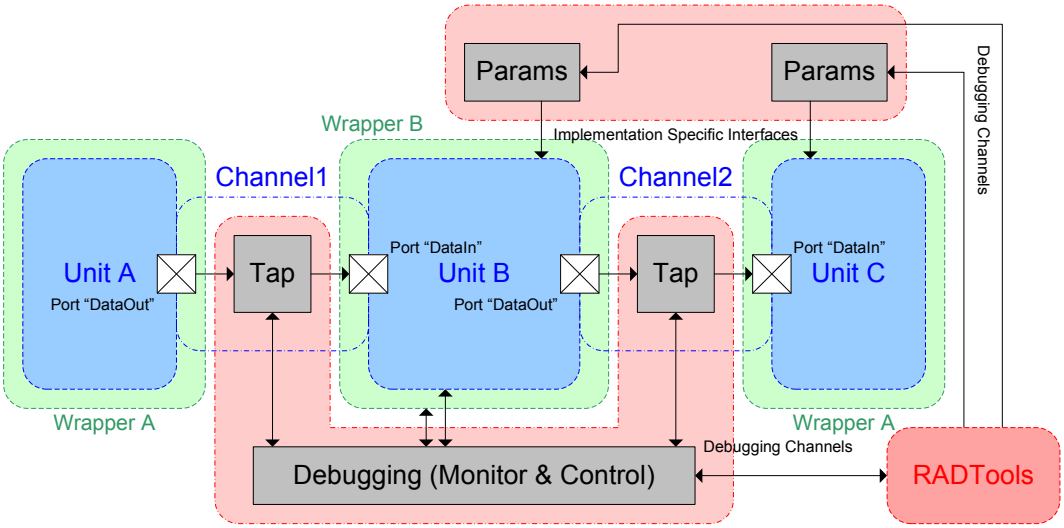
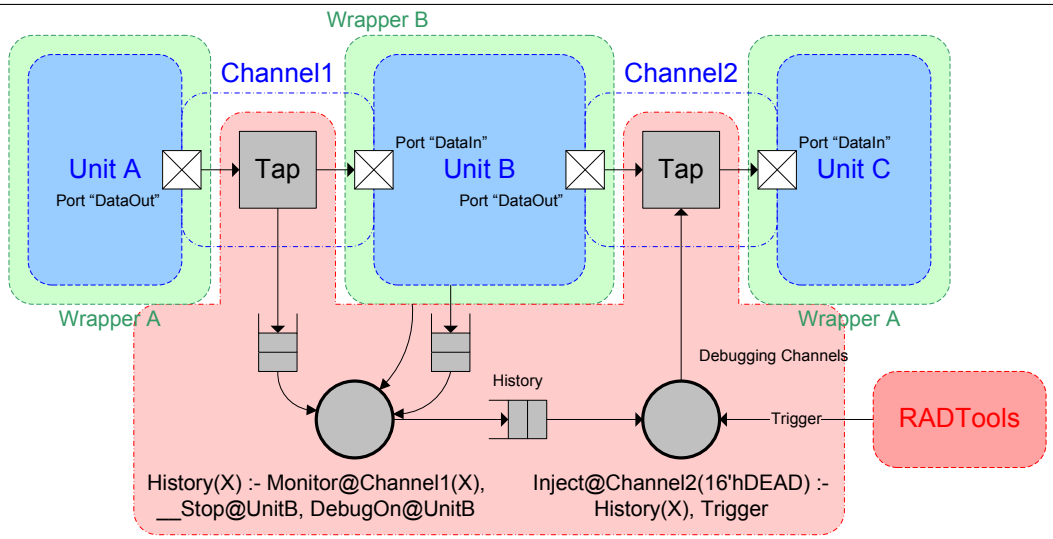


Figure 69 P2 Debugging Example



Chapter 17

Acknowledgements

The author would like to thank John Wawrzynek, Krste Asanović and Andrew Schultz who helped form the ideas which this thesis presents, primarily the original models which underly this work. The author would also like to thank Nathan Burkhart who worked on the P2 and RADTools projects, Lilia Gutnik who worked on the RADTools project, Ivan Sutherland, Igor Benko and Adam Megacz without whom Fleet would not be what it was, and Alex Krasnov and Jue Sun who did the hard work of porting RAMP Blue to RDL.

Of course generous thanks go the rest of the RAMP Gateware, RAMP Blue, RAMP Gold and RAMP Undergrads groups at UC Berkeley, including Heidi Pan, Zhangxi Tan, Tracy Wang, Ilia Lebedev and Chris Fletcher, for their suggestions and contributions, as well as those who took the time to read and comment on the original technical report, in particular Dave Patterson and Derek Chiou who commented early and often. Many thanks also go to the RDL Bootcamp attendees from March 2007 including Andrew Putnam, Eric Chung and Hari Angepat, the entire RAMP group, and anyone else who provided feedback on, worked with or complained about RDL. A special thanks goes to Dan Burke for his immoral support; he always has a good point.

The author would like to acknowledge the support of the students, faculty and sponsors of the Berkeley Wireless Research Center. This material is based upon work supported by the National Science Foundation under Grant Nos. 0403427 and 0551739. The author acknowledges the strong support of the Gigascale Systems Research Center (GSRC) Focus Center, one of five research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation (SRC) program.

Finally, and most importantly I would like to thank my parents Jeff and Marsha Gibeling, and the person who made this possible: Stella Abad.

Appendix A

References

[1] Eclipse. URL: http://www.eclipse.org.

[2] Eclipse Bug 163680. URL: https://bugs.eclipse.org/bugs/show_bug.cgi?id=163680.

[3] Java 5 (J2SE). URL: http://java.sun.com/j2se/1.5.0/.

[4] Java 6 (J2SE). URL: http://java.sun.com/j2se/1.6.0/.

[5] Javadoc Tool. URL: http://java.sun.com/j2se/javadoc/.

[6] JCraft. URL: http://www.jcraft.com/.

[7] JSCH. URL: http://www.jcraft.com/jsch/.

[8] RADS Class Fall 2006. URL: http://radlab.cs.berkeley.edu/wiki/RADSClassFall06.

[9] RAMP: Research Accelerator for Multiple Processors. URL: http://ramp.eecs.berkeley.edu.

[10] SimpleScalar. URL: http://www.simplescalar.com/.

[11] Sun Microsystems. URL: http://www.sun.com.

[12] Sun Microsystems, Java. URL: http://java.sun.com.

[13] Swing Concurrency Tutorial. URL: http://java.sun.com/docs/books/tutorial/uiswing/concurrency/index.html.

[14] Virtutech Simics. URL: https://www.simics.net/.

[15] Xilinx ISE Foundation. URL: http://www.xilinx.com/ise/logic_design_prod/foundation.htm.

[16] Stephen Robert Adams. *Modular Grammars for Programming Language Prototyping*. PhD thesis, University of Southampton, 1991.

[17] Altera. Quartus II Help Version 7.2, 2007.

[18] Altera. Altera's DE2 Development and Education Board, 2008. URL: http://university.altera.com/materials/boards/unv-de2-board.html.

[19] ARM. AMBA AXI Protocol v1.0 Specification, 2004.

[20] D. Behrens, K. Harbich, and E. Barke. Hierarchical partitioning. 1996. 1996 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No.96CB35991). IEEE Comput. Soc. Press. 1996, pp. 470-7. Los Alamitos, CA, USA.

[21] V. Betz and J. Rose. VPR: a new packing, placement and routing tool for FPGA research. In *Field-Programmable Logic and Applications. 7th International Workshop, FPL '97. Proceedings. London, UK. 1-3 Sept. 1997.*, 1997.

[22] Nathan L. Binkert, Erik G. Hallnor, and Steven K. Reinhardt. Network-Oriented Full-System Simulation using M5, 2003.

[23] Kevin Camera, Hayden Kwok-Hay So, and Robert W. Brodersen. An Integrated Debugging Environment for Reprogrammble Hardware Systems, 2005.

[24] Kevin Brandon Camera. *Efficient Programming of Reconfigurable Hardware through Direct Verification*. PhD thesis, UC Berkeley, 2008.

[25] C. Chang, J. Wawrzynek, and R. W. Brodersen. BEE2: a high-end reconfigurable computing system. *IEEE Design & Test of Computers*, 22(2):114–25,

2005. Publisher: IEEE, USA. URL: http://portal.acm.org/citation.cfm?id=1058221.1058286\&coll=GUIDE\&dl=GUIDE.

[26] Chen Chang. BEE3 Pricing & Availability, 2008. URL: http://ramp.eecs.berkeley.edu/Publications/BEE3%20Pricing%20&%20Availability%20(Slides,%201-17-2008).pdf.

[27] Ou Chao-Wei and S. Ranka. Parallel incremental graph partitioning using linear programming. In *Proceedings of Supercomputing '94. Washington, DC*, 1994.

[28] Eric S. Chung, Eriko Nurvitadhi, James C. Hoe, Babak Falsafi, and Ken Mai. ProtoFlex: FPGA-accelerated Hybrid Functional Simulation, 2007.

[29] C. Cifuentes and S. Sendall. Specifying the semantics of machine instructions. In *Proceedings 6th International Workshop on Program Comprehension. IWPC'98. Ischia, Italy. IEEE Comput. Soc. Tech. Council on Software Eng. 24-26 June 1998*, 1998.

[30] Cristina Cifuentes, Mike Van Emmerik, Norman Ramsey, and Brian Lewis. Experience in the Design, Implementation and Use of a Retargetable Static Binary Translation Framework, 2002.

[31] W. S. Coates, J. K. Lexau, I. W. Jones, S. M. Fairbanks, and I. E. Sutherland. FLEET-zero: an asynchronous switching experiment. In *Proceedings Seventh International Symposium on Asynchronous Circuits and Systems. ASYNC 2001. Salt Lake City, UT*, 2001. Proceedings Seventh International Symposium on Asynchronous Circuits and Systems. ASYNC 2001. IEEE Comput. Soc. 2001, pp.173-82. Los Alamitos, CA, USA. Size 3.5E-07 m.

[32] Jason Cong and MLissa Smith. A Parallel Bottom-up Applications to Circuit Clustering Algorithm with Partitioning in VLSI Design, 1993.

[33] John Connors, Ferenc Kovac, and Greg Gibeling. CaLinx2 EECS 15x FPGA Lab Board Technical Manual, 2004.

[34] J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In *2006 Design Automation Conference. San Francisco, CA*, 2006. 2006 Design Automation Conference (IEEE Cat. No. 06CH37797) . IEEE. 2006, pp. 657-62. Piscataway, NJ, USA.

[35] Jordi Cortadella and Mike Kishinevsky. Synchronous Elastic Circuits with Early Evaluation and Token Counterflow, 2007.

[36] D. E. Culler and Arvind. Resource requirements of dataflow programs. In *Honolulu, HI*, 1988. 15th Annual International Symposium on Computer Architecture. Conference Proceedings (Cat. No.88CH2545-2). IEEE Comput. Soc. Press. 1988, pp.141-50. Washington, DC, USA.

[37] Pierre-Yves Droz. *Physical Design and Implementation of BEE2: A High End Reconfigurable Computer*. PhD thesis, UC Berkeley, 2005. URL: http://bee2.eecs.berkeley.edu/papers/BEE2_Droz_MS_report_v2.pdf.

[38] Joel Emer, Michael Adler, Artur Klauser, Angshuman Parashar, Michael Pellauer, and Murali Vijayaraghavan. Hasim, 2007.

[39] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. 1982. ACM IEEE Nineteenth Design Automation Conference Proceedings . IEEE. 1982, pp. 174-81. New York, NY, USA.

[40] C. A. Fields. Creating hierarchy in HDL-based high density FGPA design. In *Proceedings of EURO-DAC. European Design Automation Conference. Brighton, UK. Gesellschaft fur Inf. e.V.. IEEE Comput. Soc. Tech. Committee for Design Autom.. IEEE Circuits & Syst. Soc.. ACM SIGDA. IFIP 10.5. EDAC. CEPIS. Eur. C.A.D. Standardization Initiative. 18-22 Sept. 1995*, 1995.

[41] Armando Fox, Michael Jordan, Randy Katz, David Patterson, Scott Shenker, and Ion Stoica. RADLab Technical Vision, 2005. URL: http://radlab.cs.berkeley.edu/w/uploads/2/23/RADLabWhite.pdf.

[42] Greg Gibeling. Advanced ResearchIndex Load. URL: http://radlab.cs.berkeley.edu/wiki/Advanced_ResearchIndex_Load.

[43] Greg Gibeling. 1st Class Instructions for FLEET, 10/4/2006 2006.

[44] Greg Gibeling. Levels of Design, 2007.

[45] Greg Gibeling. Levels of Design: Examples, 2007.

[46] Greg Gibeling. File Templates, 2008.

[47] Greg Gibeling, Krste Asanovic, Chris Batten, Rose Liu, and Heidi Pan. Generalized Architecture Research: An Application Server, 2008.

[48] Greg Gibeling, Andrew Schultz, John Wawrzynek, and Krste Asanovic. RAMP Architecture, Language and Compiler, 2007.

[49] M. Gschwind. Instruction set selection for ASIP design. In *Proceedings of the International Conference on Hardware and Software. Rome, Italy. ACM SIGDA. IEEE Comput. Soc.. ACM SOGSOFT. IFIP WG 10.5. 3-5 May 1999*, 1999.

[50] M. Gschwind and E. Altman. Precise exception semantics in dynamic compilation. In R. N. Horspool, editor, *Compiler Construction. 11th International Conference, CC 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002. Proceedings. Grenoble, France. 8-12 April 2002*, 2002.

[51] So Hayden Kwok-Hay and Brodersen Robert. A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. *Trans. on Embedded Computing Sys.*, 7(2):1–28, 2008. 1331338.

[52] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–77, 1978. USA.

[53] So Hoyden Kwok-Hay and R. W. Brodersen. Improving usability of FPGA-based reconfigurable computers through operating system support. In *2006 International Conference on Field Programmable Logic and Applications. Madrid, Spain. 28-30 Aug. 2006*, 2006. 2006 International Conference on Field Programmable Logic and Applications (IEEE Cat. No. 06EX1349). IEEE. 2006, pp. 349-54. Piscataway, NJ, USA.

[54] Scott E. Hudson, Frank Flannery, C. Scott Ananian, Dan Wang, and Michael Petter. CUP User's Manual, 2006. URL: http://www2.cs.tum.edu/projects/cup/manual.html.

[55] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the Internet with PIER, 2003.

[56] BlueSpec Inc. BlueSpec Overview, 2005.

[57] K. Keutzer. DAGON: technology binding and local optimization by DAG matching. In *24th ACM/IEEE Design Automation Conference Proceedings 1987. Miami Beach, FL*, 1987.

[58] J. Kilter and E. Barke. Architecture driven partitioning. In W. Nebel and A. Jerraya, editors, *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001. Munich, Germany. EDAA. EDAC. IEEE-CS TTTC. IEEE-CS DATC. ECSI. RAS Russian Acad. Sci.. IPPM. ACM-SIGDA. IFIP 10.5. AEIA. ATI. CLRC. CNR. Estonian E Soc.. GI. GMM. HTE. ITG. KVIV. VDE. 13-16 March 2001*, 2001.

[59] S. Kirkpatrick, Jr. C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598), May 1983.

[60] Gerwin Klein. JFlex Users Manual, 2005. URL: http://jflex.de/manual.pdf.

[61] E. Kohler, R. Morris, and Chen Benjie. Programming language optimizations for modular router configurations. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems. San Jose, CA*, 2002. ACM. Sigplan Notices (Acm Special Interest Group on Programming Languages), vol.37, no.10, Oct. 2002, pp.251-63. USA.

[62] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–97, 2000. Publisher: ACM, USA. URL: http://citeseer.ist.psu.edu/320570.html.

[63] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P. Y. Droz. Ramp blue: a message-passing manycore system in FPGAs. In *2007 International Conference on Field Programmable Logic and Applications, FPL 2007. Amsterdam, Netherlands. 27-29 Aug. 2007*, 2007. 2007 International Conference on Field Programmable Logic and Applications, FPL 2007. IEEE. pp. 54-61. Piscataway, NJ, USA. URL: http://ramp.eecs.berkeley.edu/Publications/RAMP%20Blue%20FPL%202007.pdf.

[64] Alex Krasnov. *RAMP Blue: A Message-Passing Manycore System as a Design Driver*. Report, UC Berkeley, 2008.

[65] H. Krupnova, A. Abbara, and G. Saucier. A hierarchy-driven FPGA partitioning method. In *Proceedings of 34th Design Automation Conference. Anaheim, CA*, 1997.

[66] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995. USA.

[67] Boon Thau Loo, Joseph M. Hellerstein, and Ion Stoica. Customizable Routing with Declarative Queries. In *Third Workshop on Hot Topics in Networks (HotNets-III)*, 2004.

[68] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: extensible routing with declarative queries. *SIGCOMM Comput. Commun. Rev.*, 35(4):289–300, 2005. 1080126. URL: http://portal.acm.org/citation.cfm?id=1080091.1080126.

[69] Boon Thau Loo, Petros Maniatis, Tyson Condie, Timothy Roscoe, Joseph M. Hellerstein, and Ion Stoica. Implementing Declarative Overlays, 2005. URL: http://portal.acm.org/citation.cfm?id=1095809.1095818.

[70] Vachharajani Manish, N. Vachharajani, and D. I. August. The Liberty structural specification language: a high-level modeling language for component reuse. In *2004 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04). Washington, DC*, 2004. ACM. Sigplan Notices (Acm Special Interest Group on Programming Languages), vol.39, no.6, June 2004, pp.195-206. USA.

[71] Larry McMurchie and Carl Ebeling. PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs, 1995.

[72] Sun Microsystems. Java Native Interface Developer Guide 6.0, 2006. URL: http://java.sun.com/javase/6/docs/technotes/guides/jni/.

[73] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Programmers, Raleigh, North Carolina, 2007. URL: http://www.pragprog.com/titles/tpantlr/the-definitive-antlr-reference.

[74] David Patterson. Research Accelerator for Multiprocessing, 2006. URL: http://ramp.eecs.berkeley.edu/Publications/RAMP8.1.ppt.

[75] David Patterson, Mark Oskin, Krste Asanovic, John Wawrzynek, Derek Chiou, James Hoe, and Christos Kozyrakis. Research Accelerator for Multiple Processors, 1/10/2007 2007. URL: http://ramp.eecs.berkeley.edu/Publications/ResearchAcceleratorforMultipleProcessors(Slides,1-10-2007).ppt.

[76] N. Ramsey and M. F. Fernandez. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524, 1997. Publisher: ACM, USA.

[77] B. M. Riess and A. A. Schoene. Architecture driven k-way partitioning for multichip modules. In *Proceedings the European Design and Test Conference. ED&TC 1995 . Paris, France. IEEE Comput. Soc.. EDA Assoc.. Eur. Group of TTC & the DATC. ACM/SIGDA. 6-9 March 1995*, 1995.

[78] A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and A. Vahdat. MACEDON: methodology for automatically creating, evaluating, and designing overlay networks. In *First Symposium on Networked Systems Design and Implementation (NSDI '04). San Francisco, CA*, 2004. First Symposium on Networked Systems Design and Implementation (NSDI '04). USENIX Assoc. 2004, pp.267-80. Berkeley, CA, USA.

[79] Andrew Schultz. RAMP Blue Design and Implementation of a Message Passing Multiprocessor System on the BEE2, 2006.

[80] Atul Singh, Petros Maniatis, Timothy Roscoe, and Peter Druschel. Distributed Monitoring and Forensics in Overlay Networks. In *Conference of the European Professional Society for Systems*, Leuven, Belgium, 2006.

[81] Hayden Kwok-Hay So and Robert W. Brodersen. *BORPH: An Operating System for FPGA-Based Reconfigurable Computers*. PhD thesis, EECS Department, University of California, Berkeley, 2007. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-92.html.

[82] IEEE Computer Society. IEEE Standard for Verilog Hardware Description Language, 2005.

[83] I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup service for Internet applications. In *ACM-SIGCOMM 2001 Conference. Applications, Technologies, Architectures, and Protocols for Computer Communications. San Diego, CA*, 2001. ACM. Computer Communication Review, vol.31, no.4, Oct. 2001, pp.149-60. USA.

[84] Jue Sun. RAMP Blue in RDL. Master's thesis, UC Berkeley, 2007.

[85] Ivan Sutherland. FLEET A One-Instruction Computer, August 25, 2005 2005.

[86] Ivan Sutherland. Four Views of FLEET, November 2, 2005 2005.

[87] Ivan Sutherland. FLEET: A One-Instruction Computer, 2006. URL: http://research.cs.berkeley.edu/class/fleet/.

[88] Benjamin Szekely and Elias Torres. A Paxon Evaluation of P2, 2005.

[89] Chuck Thacker and John Davis. BEE3 Update, 2008. URL: http://ramp.eecs.berkeley.edu/Publications/BEE3%20Update%20(Slides,%203-2-2008).ppt.

[90] John Wawrzynek, Arvind, Krste Asanovic, Derek Chiou, James C. Hoe, Christoforos Kozyrakis, Shih-Lien Lu, Mark Oskin, David Patterson, and Jan Rabaey. RAMP Research Accelerator for Multiple Processors, 2006. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-158.pdf.

[91] E. Witchel and M. Rosenblum. Embra: fast and flexible machine simulation. In *1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems. Philadelphia, PA*, 1996.

[92] Xilinx. Development System Reference Guide, 2008.

[93] Xilinx. Virtex-5 LXT ML505 Evaluation Platform, 2008. URL: http://www.xilinx.com/products/devkits/HW-V5-ML505-UNI-G.htm.

[94] Xilinx. Xilinx XUP Virtex-II Pro Development System, 2008. URL: http://www.xilinx.com/univ/xupv2p.html.

[95] Cheon Yongseok and D. F. Wong. Design hierarchy guided multilevel circuit partitioning. 2002. Proceedings of ISPD'02. 2002 International Symposium on Physical Design. ACM. 2002, pp. 30-5. New York, NY, USA.

[96] Jin Yujia, N. Satish, K. Ravindran, and K. Keutzer. An automated exploration framework for FPGA-based soft multiprocessor systems. In *International Conference on Hardware/Software Codesign and System Synthesis. Jersey City, NJ*, 2005.

Appendix B

Glossary

Application A software program which runs on a target system to do, in the real-world sense, useful work. Benchmarks, commercial workloads, and even simple demonstration programs are all applications.. 2, 7, 19, 91, 100, 137, 149, 152, 154, 155

Application Server The front end software component responsible for experimental setup and parameterization, time sharing of the back end and experimental portability across implementations. The application server may include components to deal with command line arguments, a back-end driver to communicate with the back end, load ELF files, create memory images, *etc.*and a proxy kernel. Please see [47] for more information about application servers.. 99, 141, 149, 151, 152

ASIC Application Specific Integrated Circuit. An integrated circuit design to perform one set of operations, generally customized to perform them efficiently. This is in contrast to FPGAs.. 2, 8, 18, 44, 102, 113, 141, 149, 152, 154, 155

Assembly A relatively simple representational transformation which recombines fragments in to a message and the the opposite of fragmentation (see Section 3.3). Assembly is implemented in the wrapper at the receiving port (see Section 4.2.3).. 20, 80, 149, 152

AST A tree data structure representing the abstract structure of a piece of source code, in this work generally RDL descriptions. Constructing an AST is the job of a compiler front end, whereas generating output from it is the job of the back end. ASTs mirror document object models in non-programming languages like HTML, XML or Microsoft Word, providing the same abstraction and potential for programmatic document (program) generation.. 44, 45, 69–72, 74, 81, 82, 112, 139, 149

Back End Comprised of the host and target, the back end is active portion of an experiment in contrast with the front end. For RAMP experiments or systems, the back end will generally consist of an FPGA-based host which has been programmed with some implementation of a target system. "Back end" also refers to the code generation and post-compilation portions of a toolflow, in particular all hardware synthesis tools and software compilers are considered part of the RDL "back end.". 24, 44, 45, 51, 55, 62, 64, 65, 69, 75, 76, 79, 81–84, 88, 89, 120, 141, 149, 151–154

Back-End Driver A plugin (dynamic link library, *etc.*) of the Application Server which enables communication between the front end and back end. Most back-end drivers will essentially be software implementations of cross-platform links.. 149, 151, 153

Behavioral Model A simulator or emulator whose behavior matches that of the system being modeled, but whose structure is quite different, in contrast to a structural model. Note that a particular emulator (or simulator) can be part structural and part behavioral, and this is a broad and continuous spectrum, rather than a simple binary classification. For example a software model of a processor may be structural at the high level (memory, processor and network) but behavioral at the low level (ALU, Register File).. 3, 38, 138, 149, 155

Bitwidth The width of a channel specified in bits. This is the width of the fragments the channel carries (and in to which the messages must be fragmented), at a rate of zero or one fragments per target cycle. Minimum bitwidth is 0, denoting a channel which carries no data, but whose messages indicate some form of timing information.. 8–10, 14, 20, 102, 149, 152

Channel The abstraction of inter-unit communi-

cation in a target system. Each channel connects exactly two units, and provides in-order, loss-less message transport at the rate of zero or one fragments per target cycle. Channels have a number of characteristics, including message type, bitwidth, forward latency (for data), buffering and backward latency (for flow control) (see Section 3.3).. 1, 4, 7–15, 17, 19–24, 26, 30, 31, 33–35, 37, 39–44, 46, 49–52, 55, 57, 64, 66, 70, 74, 76, 83, 86, 87, 101–110, 112, 115, 118, 124, 125, 127, 129, 131, 132, 135–138, 140, 149, 151–155

Emulation A functional only emulation of a target system, in contrast to a simulation which provides target cycle accuracy. Emulation will provide much higher performance, as it removes the overhead of detailed simulation time-keeping, and may be viewed as a instance of the target system (rather than a model) when a structural model style of modeling is used. The ability to switch between emulation and simulation is key to the application development process.. 1, 4, 7, 11–13, 15, 26, 42, 46, 59, 64, 99, 115, 124, 137, 149, 151–153, 155

Engine The firmware module or software object responsible for driving the simulation or emulation. In hardware, this translates to generating clock, reset and possible clock-valid (locked) signals. In software an engine is tantamount to a user level thread scheduler responsible for dispatching execution requests to unit implementations, possibly including distributed execution semantics.. 17, 24–26, 55, 66, 85, 87, 149, 152

Firmware Firmware may take the form of hardware or software, and includes all those I/O blocks and drivers necessary to make a platform usable by either gateware or portable software. In particular firmware should include the engine, some elements of back end to front end communication and often a DRAM interface. In the future, a standard set of RAMP gateware may be provided and maintained as a cross-project effort.. 3, 14, 17, 25, 26, 55, 125, 134, 140, 149, 152, 155

Flit The unit of flow control or routing in a network, in constrast to a phit. *E.g.* a word (byte) on RS232, a packet over Ethernet, or a message in the target model. Note that the definition of a flit in the host model is link-dependent and may range from bits (I^2C) to packets (Ethernet).. 9, 10, 12, 15, 149, 153, 154

FPGA Field Programmable Gate Array. A form of ASIC whose application is the emulation of circuits, making it roughly an order of magnitude less efficient. The benefit of FPGAs is that they are reconfigurablew while in-circuit, allowing them to change applications quickly, a feature particularly useful for research environments requiring experimental flexibility. This is in contrast to ASICs.. i, 1–4, 8, 11–13, 17, 19, 22–25, 30, 32, 36, 37, 39, 42–44, 46, 51, 54, 55, 57, 59, 62, 64–66, 69, 74, 80, 83–85, 88, 89, 91, 99, 101–104, 106–109, 111, 112, 115, 117, 123–125, 127–132, 134, 135, 137, 140, 149, 151–155

Fragment Fragments are the unit of data transport over channels and the target level phit. Notice that while channels may carry large messages, they must be fragmented and assembled. fragmentation is one of the primary ways in which RDF enables the parameterization of performance simulations.. 9–12, 15, 20, 23, 149, 151, 152, 154, 155

Fragmentation A relatively simple representational transformation which breaks a message up in to fragments and the the opposite of assembly (see Section 3.3). Fragmentation is implemented in the wrapper at the sending port.. 10, 12, 20, 21, 80, 149, 151, 152

Front End The front end of an experiment consists of the management (see Section 11), monitoring & debug and Application Server hardware or software. For RAMP experiments or systems, the front end will generally consist of an simple x86/Linux PC running *e.g.* FPGA programming tools and perhaps an SSH server to allow remote access. "Front end" also refers to the input stage of a compiler, in particular the lexical and syntactic analysis portions of RDLC. This is in contrast to the back end.. 24, 44, 45, 65, 74, 81–83, 99, 126, 141, 149, 151, 152, 154

Gateware Portable hardware, described in some HDL reducible to gates, generally at RTL. The defining characteristic of gateware is that it can be compiled or synthesized to a variety of platforms, assuming the proper firmware is available.. 8–10, 12, 17, 19, 20, 22, 24, 37, 47, 55, 59, 66, 73, 101, 109, 115, 123–127, 129–133, 143, 149, 152, 153, 155, 156

Hardware Any kind of circuit, including all gateware, PCBs and ICs. This is in constrast to software.. 1–4, 8, 11, 12, 15, 18, 19, 22, 23, 25,

sub-message types or arrays of a single message sub-type.. 4, 7–12, 15, 19–21, 24, 27–37, 47, 49, 51, 52, 54, 56, 64, 66, 74, 76, 87, 101, 103, 118, 124, 125, 127, 128, 140, 141, 149, 151–156

NoC A packet or circuit switch network implemented entirely within an ASIC for communication. Networks may be preferable to busses in designs with higher bandwidth or larger chip area.. 12, 114, 141, 149

Outside Edge The interface between the wrapper and links as implemented in the host system. The exact details of this interface vary widely with the links the wrapper connects to, to the point where the link generator plugins (see Section 10.4.1) are responsible for dynamically specifying and implementing this interface.. 19, 26, 149, 153

Packing A representational transformation which reduces an array of one or more dimensions to a simple vector and the opposite of *unpacking*. Performed at output ports of modules which are RDL units, and exists solely to compensate for the lack of full port-array support in Verilog (and other languages). In languages like BlueSpec, Java or C this is unecessary.. 20, 21, 36, 51, 80, 128, 149

PAR Place and Route, the main component of the FPGA design compilation process. Though technically preceeded by synthesis, PAR often is used as a catch-all for the complete FPGA tool flow from HDL to device programming information (bitfile). PAR can be extremely slow for large, complex or high speed designs as placement is an NP-complete problem.. 11, 51, 65, 69, 132, 149

Phit The unit of physical transfer over a network, in contrast to a flit. *E.g.* a bit on RS232, a nibble over Ethernet MII, or a fragment in the target model. Note that the definition of a phit in the host model is link-dependant.. 9–11, 15, 149, 152

Platform A component of the back end system, on to which units may be mapped. Note that platforms may be constructed hierarchically out of smaller platforms. Thus examples of a platform would include a single FPGA, a board with multiple FPGAs, a laptop computer, and even a laptop connected to a multi-FPGA board over Ethernet.. i, 1, 3, 4, 8, 12, 14, 15, 17, 19, 22–28, 30–47, 50–55, 57, 59, 63–65, 67, 69, 74–77, 79, 82, 83, 85–87, 89, 98, 99, 101–112, 124, 125, 130, 131, 134, 137, 139–141, 149, 151–153, 155

Port The point of connection between a unit and a channel. Port characteristics are entirely static and limited to the type of messages the port will carry (which must match the type of messages carried by the port it is connected to). In implementation a port will be able to transfer at most one message per target cycle and must therefore be as wide as the largest message it can support. Ports will operate under FIFO style semantics, with, in hardware, a `__READY` signal to indicate data (on an input) and free space (on an output) along with a `__READ` (input) or `__WRITE` (on an output).. 8–10, 12, 15, 21, 24, 27, 30–41, 47, 49–52, 54, 56, 57, 63–66, 70, 74, 76, 86, 87, 115–118, 128–130, 139, 149, 151–154, 156

Proxy Kernel A proxy kernel services kernel or system calls which are redirected from the target system under test to the front end. Aside from eliminating the requirement that the target be complete enough to boot an OS, this allows applications running on the target to access the file system of the front end machine transparently. It should be noted that a virtual machine may be used to restrict the proxy kernel's ability to escape or damage the front end.. 149, 151

RAMP Research Accelerator for Multiple Processors [9]. A collaboration of a researchers at many different universities, RAMP is an umbrella project for FPGA-based architecture research rather than a single project with an expected single design artifact. Generally the projects within RAMP are limited to a subset of the overall project, and are given a color (generally a school color for the university they are centered at) to set them apart. For example RAMP Blue [79, 63] and RAMP Gold are both specific projects within RAMP which have or are taking place at U.C. Berkeley.. i, 1–5, 7, 8, 12, 14, 15, 17–19, 24, 26, 27, 32, 33, 36, 38, 39, 41, 44–46, 52, 56, 59, 61, 62, 67, 71, 77, 81, 82, 88, 89, 91, 96, 99–102, 105, 109, 110, 112, 113, 123–125, 130, 131, 133, 135–141, 143, 149, 151, 152, 154, 155

RDF RAMP Design Framework. The modeling framework within which RAMP target systems must fit (see Section 2.4). Includes design restrictions (see Section 3.6), modeling conven-

of that functionality, and should be portable gateware or software preferably parameterized, thereby enhacing their potential for composition and reuse. Note that a unit will need to be aware of the host semantics of RDF (see Section 3.4).. 1, 4, 7–15, 17, 19–31, 33–47, 49–52, 54–57, 59, 62–67, 70–77, 81–83, 85–88, 101–108, 112, 115–118, 124–130, 132, 133, 135–141, 149, 151–156

Unmarshaling A representational transformation which uncompresses an RDL message, by adding back in the meaningless bits removed by *marshaling*. This is one of the more complex transformations done by RDLC, and may or may not always be necessary.. 19, 51, 149

Unpacking A representational transformation which expands a simple vector to an array of one or more dimensions and the opposite of *packing*. Performed at input ports of modules which are RDL units, solely to compensate for the lack of full port-array support in Verilog (and other languages). In languages like BlueSpec, Java or C this is unecessary.. 80, 149

Wrapper The Verilog, Java, C or similar implementation code which forms an interface between a unit, and the remainder of the host and implementation. Wrappers are generated by RDLC and are responsible for providing a clean set of ports to a unit and hiding from it the details of the implementation.. 9, 17, 19–26, 37, 38, 59, 63–66, 71, 83, 85, 149, 151–154